PROGRAMMA
INTERNATIONAL, Inc.
3400 Wilshire Boulevard
Los Angeles, CA 90010

(213) 384-0579

PROGRAMMA

A S M / 6 5 E D I T O R A S S E M B L E R

REFERENCE MANUAL

VER 1.Ø

JUNE 1979

COPYRIGHT (c) 1979 BY COMPASS MICROSYSTEMS & PROGRAMMA INTL., INC.
ALL RIGHTS RESERVED. Reproduction in part or form of the contents of this document or its accompanying cassette tape or disk, except for the personal use of the original purchaser, is strictly forbidden without the expressed written consent and permission of PROGRAMMA International, Inc.

COPYRIGHT (c) 1979 BY PROGRAMMA INTERNATIONAL, INC.

JUNE 1979 EDITION

This edition (2200.001) is a major revision and obsoletes all previous editions and documents.

Technical changes are marked with a bar in the outer margin. Changes due to subsequent releases will be documented in the future publication bulletins or revisions.

Requests for copies of PROGRAMMA publications should be made to your PROGRAMMA representative or to the PROGRAMMA central office.

A reader's comment form is provided at the back of this publication. If the form has been removed, comments may be addressed to:

PROGRAMMA International, Inc.
Publications Department
P.O. Box 70279
Los Angeles, CA 90070

P R E F A C E

This reference publication is intended for programmers using the PROGRAMMA International, Inc. ASM/65 Editor Assembler System. This publication describes how to write assembler source statements, system start-up, alteration of system parameters, and error handling.

The machine instructions are summarized in Appendix E. Machine instructions and extended mnemonic codes are summarized in Section 3, and additional reference tables appear in Appendixes B, C, and D. The machine instructions are described in detail in the publication "MCS6500 MICROCOMPUTER FAMILY PROGRAMMING MANUAL" by MOS Technology, Inc.

Description of the PIE Text Editor included in the ASM/65 Editor Assembler System are found in the attached PIE Text Editor manual accompanying this document.

The reader should be familiar with the hardware manuals and operational procedures of the devices attached to his particular computer configuration.

TABLE OF CONTENTS

SECTION 1. INTRODUCTION

FUNCTION OF THE ASSEMBLER..... 1.1

RELATIONSHIP TO THE OPERATING SYSTEM..... 1.2

SYSTEM REQUIREMENTS..... 1.2

SECTION 2. WRITING SOURCE STATEMENTS

CHARACTER SET..... 2.1

BASIC FORMAT OF SOURCE STATEMENTS..... 2.4

TERMS AND EXPRESSIONS

TERMS..... 2.5

CONSTANTS..... 2.6

STRING CONSTANTS

CHARACTER STRING CONSTANT... 2.6

ARITHMETIC CONSTANTS

DECIMAL CONSTANT..... 2.8

HEXADECIMAL CONSTANT..... 2.8

OCTAL CONSTANT..... 2.8

BINARY CONSTANT..... 2.8

SYMBOLS

ORDINARY SYMBOLS..... 2.10

ASSEMBLER SYMBOLS..... 2.10

LOCATION COUNTER REFERENCE..... 2.11

LITERALS..... 2.11

EXPRESSIONS AND OPERATORS..... 2.12

EVALUATION OF EXPRESSIONS..... 2.13

CODING FORM..... 2.13
NAME FIELD..... 2.13
OPERATION FIELD..... 2.15
OPERAND FIELD..... 2.15
COMMENT FIELD..... 2.15
IDENTIFICATION-SEQUENCE FIELD..... 2.15

SECTION 3. MACHINE INSTRUCTIONS

SOURCE STATEMENTS
OPERANDS..... 3.1
NAME ATTRIBUTE..... 3.1
SUMMARY OF MACHINE INSTRUCTIONS..... 3.2
ADDRESSING MODES
ABSOLUTE ADDRESSING..... 3.6
PAGE ZERO ADDRESSING..... 3.7
IMMEDIATE ADDRESSING..... 3.7
IMPLIED ADDRESSING..... 3.8
ACCUMULATOR ADDRESSING..... 3.8
RELATIVE ADDRESSING..... 3.8
INDEXED ADDRESSING..... 3.9
INDIRECT ADDRESSING..... 3.9

SECTION 4. ASSEMBLER INSTRUCTIONS

SOURCE STATEMENT OVERVIEW..... 4.1
PROGRAM CONTROL STATEMENTS
.END - END ASSEMBLY..... 4.2
.FILE - CHAIN FILES..... 4.3
.OPT - OPTIONS..... 4.4
.OFS - OFFSET..... 4.6

SYMBOL AND DATA DEFINITION STATEMENTS

*= - SET LOCATION COUNTER..... 4.7

= - EQUATE..... 4.8

.BYT - BYTE DEFINED DATA..... 4.9

.WOR - WORD DEFINED ADDRESS DATA..... 4.10

.DBY - WORD DEFINED DATA..... 4.10

LISTING CONTROL STATEMENTS

.PAGE - IDENTIFY LISTING/NEW PAGE..... 4.11

.SKIP - INSERT BLANK LINES..... 4.12

SECTION 5. USING THE ASSEMBLER

LOADING THE ASSEMBLER..... 5.1

FUNCTION MENU

ASSEMBLE..... 5.2

CATALOG..... 5.3

EDIT..... 5.4

LOAD..... 5.4

PRINT..... 5.4

QUIT..... 5.4

RE-EDIT..... 5.5

SAVE..... 5.5

THE RESET KEY..... 5.5

SECTION 6. MEMORY MAP

SYSTEM MEMORY MAP..... 6.1

APPENDIX A. ASCII REPRESENTATION..... A.1
APPENDIX B. PROGRAMMING MODEL MCS650X..... B.1
APPENDIX C. ALPHABETICAL LIST OF MNEMONICS..... C.1
APPENDIX D. HEX CODE TO MNEMONIC CODE..... D.1
APPENDIX E. SUMMARY OF ASSEMBLER STATEMENTS..... E.1
APPENDIX F. ASSEMBLER ERROR MESSAGES..... F.1

1. INTRODUCTION

FUNCTION OF THE ASSEMBLER

The PROGRAMMA ASM/65 Editor Assembler consists of a language and an assembler program. The language is a set of codes and coding rules for writing a source program. The process of translating a mnemonic or symbolic form of a computer program (source code) to actual processor instructions (object code) is called an assembly, and a program which performs the translation is called an assembler. The assembler program translates the source program into an object program that can be executed by the system. The object program produced by the assembler is in the form of an absolute object module.

Assemblers which are designed to run on a 6500-based system are referred to as resident assemblers. ASM/65 is a two pass assembler which is a compatible subset of the FORTRAN cross-assemblers available for the 6500 Series family. Source code is prepared using the PROGRAMMA PIE Text Editor, and then either stored on the disk or passed directly to the ASM/65 Assembler.

Machine instructions and assembler instructions are the two types of source statements that can be expressed in the assembler language.

The machine instruction source statements provide mnemonic codes for all machine instructions in the MOS Technology 6500 Series Microprocessor instruction set. Extended mnemonics are also provided for the branch instructions. A complete description of the machine instructions, addressing techniques, and data representation is in the manual "MCS6500 MICROCOMPUTER FAMILY PROGRAMMING MANUAL" by MOS Technology, Inc.

The assembler instruction source statements specify auxiliary functions to be performed by the assembler program. These functions include:

Checking and documenting programs

Controlling address assignment

Defining data and symbols

Controlling the assembly process

A listing of the source program statements and the resulting object program statements may be produced with programmer control of form and contents. A list of symbol definition is also produced unless it

© 1976 PROGRAMMA CONSULTANTS

is suppressed by the programmer. Errors detected during an assembly are indicated on the user output list device and cannot be suppressed by the programmer.

The operation of the Assembler is completely automatic. The user, however, has control of how much of the listing is generated, where the listing is printed, and whether the complete text of ASCII strings are printed. The Assembler also has a powerful file linkage which allows multiple source files to be assembled at one time.

RELATIONSHIP TO THE OPERATING SYSTEM

The ASM/65 Editor Assembler program is an independent component of the APPLE II DOS 3.2 Disk Operating System and operates under its control for I/O service only. The APPLE II ROM Monitor provides the ASM/65 Editor Assembler program with input/output services needed for its proper functioning.

SYSTEM REQUIREMENTS

The APPLE II System equipment configuration required to execute the assembler program is as follows.

48K bytes of main storage

One source input device (Diskette or keyboard)

One list device (CRT)

One input device (Keyboard)

One storage device (Diskette)

One optional print device (Printer)

© 1976 PROGRAMMA CONSULTANTS

2. WRITING SOURCE STATEMENTS

To write source statements, the programmer should be familiar with the following topics:

Character set

Basic format of source statements

Types of terms and expressions

Coding form

CHARACTER SET

Source statements may contain the following characters:

LETTERS	A through Z
DIGITS	Ø through 9
SPECIAL CHARACTERS	+ * & - (; ,) " . ' = / # @ < > blank

The ASCII format for these coded characters are listed in Appendix A. Any of the 127 combinations may appear inside a character constant or in a comments statement. The meaning of these characters, and combinations of these characters, are explained in Figure 2-1.

© 1976 PROGRAMMA CONSULTANTS

CHARACTER	EXPLANATION	EXAMPLE
A through Z	Used in symbol and character string constants	LDA #'D
Ø through 9	Used in numeric constants and symbols	LDA #5
,	Operand or suboperand separator	.BYT $10,10
$	Defines a numeric constant in base 16 (Hexadecimal)	LDA $AB10
@	Defines a numeric constant in base 8 (Octal)	.BYT @176
%	Defines a numeric constant in base 2 (Binary)	.BYT %01101
'	Defines an ASCII literal constant	.BYT '&'
#	Indicates a literal term or immediate addressing mode	LDA #'X
*	Location counter reference indicator	LDA *
+	Addition arithmetic operator indicator	LDA %101+7
-	Subtraction arithmetic operator indicator	LDA $40-7
>	High byte selection indicator	LDA #>HIGH
<	Low byte selection indicator	LDA #<LOW

FIGURE 2-1. CHARACTER USAGE

© 1976 PROGRAMMA CONSULTANTS

CHARACTER	EXPLANATION	EXAMPLE
=	Assigns the value of an expression (equate)	INADDR =$02
.	Preceeds all assembler pseudo operators	.BYT 'ABC'
;	Begins a comment or remark information	; TITLE PAGE
A	Implies Accumulator addressing mode	ASL A
X	Implies Indexed addressing mode	DEC ARRAY,X
Y	Implies Indexed addressing mode	LDA INADDR,Y
( )	Implies Indirect addressing mode	LDA (CURSOR),Y
blank	Field separator	LDA $BA

FIGURE 2.1 CONTINUATION. CHARACTER USAGE

© 1976 PROGRAMMA CONSULTANTS

BASIC FORMAT OF SOURCE STATEMENTS

Source program statements have the fields outlined in Figure 2-2.

NAME	OPERATION	OPERAND	COMMENT
Any symbol or blank	Machine instruction assembler instruction	Single expression, several expressions, or blank	;informational material or blank

FIGURE 2-2. SOURCE STATEMENT FORMAT

The name field entry is a symbol used to identify a statement. The name field is necessary for certain statements, or when the statement is referred to in another statement, such as in a Jump instruction.

The operation field entry is a predefined mnemonic code (or mnemonic) which identifies the function of a machine or assembler instruction. Mnemonics are designed to be easily learned and remembered; for example, LDA for Load Accumulator.

The operand field entry defines or identifies the data involved in the operation. Most statements have one or two operands, although some statements have no operands at all. Each operand has one or more terms, which may be used in a combination to form one or more expressions. (Refer to immediately following text for a discussion of terms and expressions.) Operands of machine statements generally represent storage locations, general registers, immediate data, or constant values. Operands of assembler statements provide the information necessary for the assembler to perform the designated operation.

The optional comment field contains any informational material the programmer wishes to add. Note that the comment field should be preceeded by a semicolon.

TERMS AND EXPRESSIONS

A term is a symbol, character, or number that represents a value; an expression is a single term or a combination of terms. An expression is used in the operand field of a source statement. The following text fully defines terms and expressions.

© 1976 PROGRAMMA CONSULTANTS

TERMS

Every term represents a value; the value may be assigned by the assembler program (symbol, symbol length attribute, location counter reference) or may be inherent in the term itself (constant, literal).

An arithmetic combination of terms is reduced to a single arithmetic value by the assembler. An arithmetic value is represented as a 16-bit binary value in two's complement form. A logical value has a range of 0 through 65,535; and an arithmetic value has a range of -32,768 through +32,767. Limitations on the value of an expression depend on its use. For example, a term requiring only 8 bits of information would be used by some instructions; a term representing an address must not exceed the size of storage.

The terms used in assembler statements are outlined in Figure 2-3. An explanation of each type of term and the rules for its use are provided in the following text.

CONSTANTS	STRING CONSTANTS	CHARACTER STRING CONSTANT	'ABCDEFGH'
	ARITHMETIC CONSTANTS	DECIMAL ARITHMETIC CONSTANT	567
		HEXADECIMAL ARITHMETIC CONSTANT	$4A
		OCTAL ARITHMETIC CONSTANT	@123
		BINARY ARITHMETIC CONSTANT	%1001001
SYMBOLS	ORDINARY SYMBOL		CURSOR
	ASSEMBLER SYMBOL		.WORD
LOCATION COUNTER			*+20
LITERALS			#62

FIGURE 2-3. TYPES OF TERMS

© 1976 PROGRAMMA CONSULTANTS

CONSTANTS

Constants are terms whose values are inherent in the terms themselves. They specify machine values or bit configurations directly, rather than by equating the values to symbols and then using symbolic references. Constants represent such program elements as immediate data, masks, addresses, and address increments.

Constants are string or arithmetic. String constants are of variable size; arithmetic constants are 16-bits long. Examples of all types of constants are presented in Figure 2-5.

STRING CONSTANTS

A string constant can only be used as a single term expression.

CHARACTER STRING CONSTANT

A character string constant is written as a string of characters enclosed in apostrophes, for example: 'STRING'. To represent the literal value of an apostrophe, a second apostrophe must immediately follow it within the string constant. The length of a character string constant is equal to the number of characters in the constant, excluding dual apostrophe instances, which do not appear in storage.

Examples of character string constants are shown in Figure 2-4.

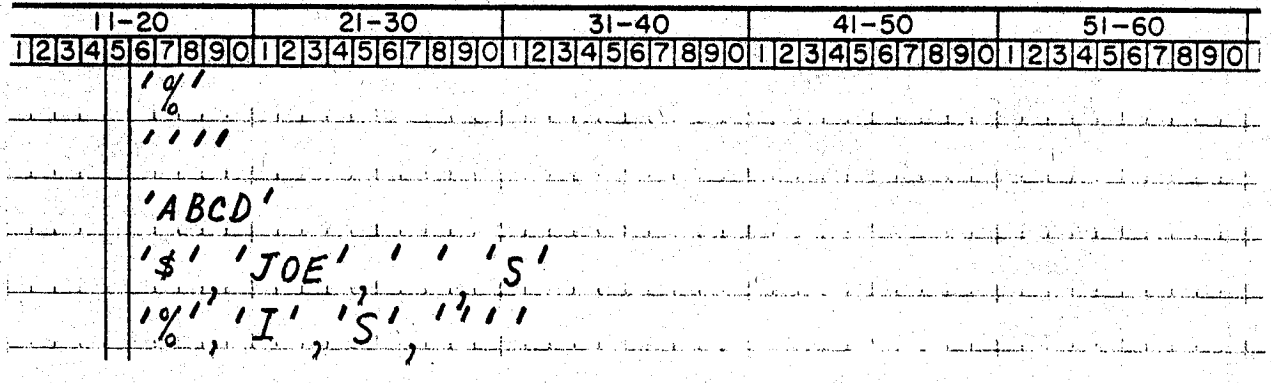


FIGURE 2-4. CHARACTER CONSTANTS

© 1976 PROGRAMMA CONSULTANTS

THIS PAGE INTENTIONALLY HAS BEEN LEFT BLANK BY US.

© 1976 PROGRAMMA CONSULTANTS

ARITHMETIC CONSTANTS

Arithmetic constants can be used in multi-term expressions. An arithmetic constant is assembled as its two-byte binary equivalent. The maximum size of an arithmetic constant is $2^{16}-1$.

DECIMAL ARITHMETIC CONSTANT

A decimal arithmetic constant is written as an unsigned integer number of 1 to 5 digits, for example: 20.

HEXADECIMAL ARITHMETIC CONSTANT

A hexadecimal arithmetic constant is written as $ followed by a string of 1 to 4 hexadecimal digits, for example: $1A4. Each hexadecimal digit is assembled as its four-bit binary equivalent.

OCTAL ARITHMETIC CONSTANT

An octal arithmetic constant is written as @ followed by a string of 1 to 5 octal digits, for example: $573.

BINARY ARITHMETIC CONSTANT

A binary arithmetic constant is written as % followed by a string of 1 to 8 binary digits, for example: %10101010.

SYMBOLS

A symbol is a character or combination of characters used to represent locations or arbitrary values. Symbols, through their use in name fields and operands, provide the programmer with an efficient way to name and reference a program element. A symbol is defined when it appears in the name field of a source statement.

© 1976 PROGRAMMA CONSULTANTS

TYPE	EXAMPLE	GENERATED HEX CODE
Character String	'ABCDEFGH' 'A3='	4142434445464748 41333D
Decimal Arithmetic	8754 123	2232 7B
Hexadecimal Arithmetic	$AB45 $D	AB45 0D
Octal Arithmetic	@16542 @364	1D62 F4
Binary Arithmetic	%00010011 %10001100	13 8C

FIGURE 2-5. EXAMPLES OF ASSEMBLED CONSTANTS

© 1976 PROGRAMMA CONSULTANTS

In general, symbols must conform to these rules:

1. The symbol must not have more than six characters.
2. The first character must be alphabetic.
3. The remaining characters may be digits, letters, or special characters.
4. The first blank after the start of a symbol terminates that symbol.
5. The 56 valid operation codes (listed in Appendix B) and the reserved symbols A, X, Y, S, and P may not be used, as each has special meaning to the assembler.

The assembler has two types of symbols: ordinary symbols and assembler symbols.

ORDINARY SYMBOLS

An ordinary symbol consists of 1 to 6 alphanumeric characters, the first of which must be a letter. Ordinary symbols identify program locations or arbitrary values.
Examples of ordinary symbols are:

BETA

ENTRY2

A$BASE

An ordinary symbol that names an instruction, a storage area, or a data definition is the address of the leftmost byte of the identified field. Address values are relocatable terms. The value of an address symbol must not exceed $2^{16}-1$.

An ordinary symbol may be defined only once in an assembly. That is, each symbol used as the name of a statement must be unique within that assembly.

ASSEMBLER SYMBOLS

An assembler symbol is a reserved word that is used as a directive to the assembler. All assembler symbols begin with a period and are used in the operation field of an instruction. Assembler symbols are further explained under the section title "ASSEMBLER INSTRUCTIONS".

© 1976 PROGRAMMA CONSULTANTS

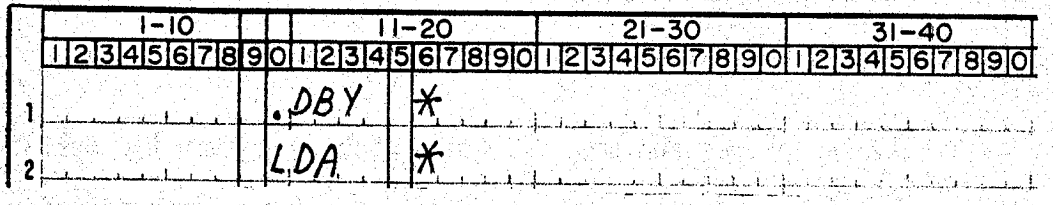
LOCATION COUNTER REFERENCE

A location counter assigns storage addresses to program statements. It is the assembler's equivalent of the instruction counter in the computer. As each machine instruction or data area is assembled, the location counter is incremented by the length of the assembled item. Thus, it always points to the next available location. If the statement is named by a symbol, the value of the symbol is the value of the location counter before addition of the length.

The maximum value for the location counter is $2^{16}-1$.

The programmer can refer to the current location counter by using an asterisk in the operand field. The asterisk represents the value of the current location counter at the start of the current statement.

An example of the use of the location counter is:



LITERALS

A literal term is used to introduce data into a program. The format of a literal term is as follows:

#a a = Data value to be generated (required); any legal expression except another literal term.

Literal terms can only be used as the operand of an immediate addressing mode instruction.

Examples of literals are shown in Figure 2-6.

```
LDA  #56       Decimal literal
LDA  #$AB      Hexadecimal literal
LDA  #@123     Octal Literal
LDA  #'A'      Character Literal
LDA  #%10111   Binary Literal
LDA  #76-9     Expression
```

FIGURE 2-6. LITERALS

© 1976 PROGRAMMA CONSULTANTS

EXPRESSIONS

An expression is defined as one or more terms linked by arithmetic operators. Expressions may be single term or multi-term.

SINGLE TERM EXPRESSION	MULTI-TERM EXPRESSION
29	SYMBOL+40
1	A+B-10
SYMBOL	10-BETA+2

During assembly, all expressions are resolved to a single value. Figure 2-7 provides an explanation of all types of operators.

The rules for coding expressions within an operand field are as follows.

1. An expression may not start with an arithmetic operator.
2. An expression may not contain two terms in succession.
3. An expression may not contain two operators in succession.
4. A multi-term expression may not contain a literal.
5. An expression cannot evaluate to negative values.

ARITHMETIC OPERATORS		
OPERATOR	MEANING	EXAMPLE
+	Addition	A+B
-	Subtraction	10-C
SPECIAL OPERATORS		
>	High-Byte	>HIGH
<	Low-Byte	<LOW

FIGURE 2-7. TYPES OF OPERATORS

© 1976 PROGRAMMA CONSULTANTS

EVALUATION OF EXPRESSIONS

A single term expression has the value of the term involved.

A multi-term expression is reduced to a single arithmetic value as follows:

1. Each term is given its value.
2. Operations are performed from left to right with no precedence specified.
3. The expression is computed to 16-bits, and it is then truncated to 8-bits depending on the contextual use.

The twos complement representation of a negative number must be expressed as an unsigned constant, for example: write a "-1" as "$FF".

CODING FORM

Figure 2-8 illustrates a source coding form provided for convenience to the programmer and the terminal operator. Since assembler statements are free form, the various fields (name, operation, operand, and comment) need not begin in any specified column. The exception to this is that the name field must always begin in column 1. The only restrictions are that the fields appear in the sequence shown, be separated by one or more spaces, and that the name field begin in column 1. If the name field is omitted, the operation field can begin in any column after column 1.

All statements are contained in columns 1 through 72. Columns 73 through 80 are reserved for identification and statement sequencing and are not used nor entered to the assembler. This makes column 1 the begin column and column 72 the end column.

Each field in the program line need only be separated by a single space. If the fields are separated in this manner, the Assembler will columnize the fields and produce a readable listing. The user's program may then be stored on the disk in a highly compressed form.

NAME FIELD

If the name field is included, it must begin in the begin column (column 1). If the name field is omitted, the begin column must be blank. A name cannot be more than six (6) characters long.

© 1976 PROGRAMMA CONSULTANTS

ASSEMBLER CODING FORM

PROGRAMMER

PAGE OF

SYSTEM NAME

SYSTEM NUMBER

CATALOGUE NUMBER

PROGRAM NAME

PROGRAM NUMBER

DATE DOCUMENTED

	NAME	OP	OPERAND	COMMENT	IDENT
	1-10	11-20	21-30	31-40 41-50 51-60 61-70	71-80
	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					

FIGURE 2-8. SOURCE CODING FORM

© 1976 PROGRAMMA CONSULTANTS

OPERATION FIELD

The operation field begins after the first blank column. More than one blank column can separate the name field and the operation field; however, within the operation field, blanks are not allowed. The operation field cannot be more than five characters.

OPERAND FIELD

Operands identify or describe data used in a statement. An operand may be actual data, a storage address, a mask, etc. A comma must separate operands when using the indexing addressing mode. Embedded blanks or spaces are not allowed, except in strings.

COMMENT FIELD

To include comments on the same line as an existing statement, the programmer must begin the comment at least one space after the operand field. A comment should be preceeded by the character semicolon. If the semicolon is omitted, the comment field will not be placed in its proper column in the listing. A semicolon starting in the begin column indicates a comment statement. Comments do not affect the assembled program, but are printed in the assembly listing.

The following example illustrates the name, operation, operand, and comments fields. LOAD ACCUMULATOR on the first line is a comment because a semicolon preceedes it and separates it from the operand. The semicolon on the second line in column 1 specifies that all characters on that line are comments.

	NAME	OP	OPERAND	COMMENT
	1-10	11-20	21-30	31-40 / 41-50
1	BEGIN	LDA	TAG,X	; LOAD ACCUMULATOR
2	;			WITH INDEXING
3				

IDENTIFICATION-SEQUENCE FIELD

The identification-sequence field is used for identification or sequencing of statements. This field is never entered into the assembler.

© 1976 PROGRAMMA CONSULTANTS

3. MACHINE INSTRUCTIONS

SOURCE STATEMENTS

Machine instruction source statements consist of:

Name field (optional)

Mnemonic operation code

Operand field

Comment field (optional)

An example of a machine instruction source statement is given below.

	NAME	OP	OPERAND	COMMENT
	1-10	11-20	21-30	31-40	41-50
	1234567890	1234567890	1234567890	1234567890	12345678
1	NAM12	LDA	AMOUNT,X	; THIS SETS AMOUNT
2				

OPERANDS

Machine instructions have 0, 1, or 2 operands. Operands are written as a single operand, or as two operands separated by a comma. The possible formats of an operand are shown below.

op

op,op

Operands specify immediate values, memory locations, or index registers. The Accumulator Addressing mode requires that the accumulator be specified as an operand.

If indexing is not desired for an instruction, the operand used for indexing is omitted.

NAME ATTRIBUTE

Any machine instruction statement may be named by a symbol, to which other assembler statements can refer. The value attribute of the symbol is the address of the leftmost byte assigned to the assembled instruction.

© 1976 PROGRAMMA CONSULTANTS

SUMMARY OF MACHINE INSTRUCTIONS

The MOS Technology 6500 Series Microprocessor machine instruction set contains 56 valid machine instruction mnemonics. These are listed and defined in Table 1-0. When assembled, each mnemonic generates one byte of machine code, the actual bit pattern depending upon both the operation specified in the opcode field and the addressing mode determined from the operand field. The operand field may generate one or two bytes of address information.

The General-Purpose instructions of the MOS Technology 6500 Series Microprocessor are divided into functional groups, and these functional groups are listed in the following table.

GENERAL PURPOSE INSTRUCTIONS

- Accumulator
- Arithmetic
- Branching
- Compare
- Data Transfer
- Index Register
- Reset & Interrupt
- Shift
- Stack Processing
- Status Register

Further information on the usage and function of the instructions in each of the functional groups can be found in the "MCS6500 Microcomputer Family Programming Manual" by MOS Technology, Inc.

The 56 valid machine instruction mnemonics are listed in alphabetical order in Table 1-0. Each mnemonic is also defined as to its general function.

© 1976 PROGRAMMA CONSULTANTS

TABLE 1-0. 6502 MICROPROCESSOR INSTRUCTION SET

	ADC	Add Memory to Accumulator with Carry
	AND	AND Memory with Accumulator
	ASL	Shift Left One Bit (Memory or Accumulator)
	BCC	Branch on Carry Clear (less than)
	BCS	Branch on Carry Set (greater than or zero)
	BEQ	Branch on Result Zero
	BIT	Test Bits in Memory with Accumulator
	BMI	Branch on Result Minus
	BNE	Branch on Result Not Zero
	BPL	Branch on Result Plus
*	BRK	Force Break
	BVC	Branch on Overflow Clear
	BVS	Branch on Overflow Set
*	CLC	Clear Carry Flag
*	CLD	Clear Decimal Mode
*	CLI	Clear Interrupt Disable Bit
*	CLV	Clear Overflow Flag
	CMP	Compare Memory and Accumulator
	CPX	Compare Memory and Index X
	CPY	Compare Memory and Index Y
	DEC	Decrement Memory by One
*	DEX	Decrement Index X by One
*	DEY	Decrement Index Y by One

© 1976 PROGRAMMA CONSULTANTS

	EOR	Exclusive-OR Memory with Accumulator
	INC	Increment Memory by One
*	INX	Increment Index X by One
*	INY	Increment Index Y by One
	JMP	Jump to New Location
	JSR	Jump to New Location Saving Return Address
	LDA	Load Accumulator with Memory
	LDX	Load Index X with Memory
	LDY	Load Index Y with Memory
	LSR	Shift Right One Bit (Memory or Accumulator)
*	NOP	No Operation
	ORA	OR Memory with Accumulator
*	PHA	Push Accumulator on Stack
*	PHP	Push Processor Status on Stack
*	PLA	Pull Accumulator from Stack
*	PLP	Pull Processor Status from Stack
	ROL	Rotate One Bit Left (Memory or Accumulator)
	ROR	Rotate One Bit Right (Memory or Accumulator)
*	RTI	Return from Interrupt
*	RTS	Return from Subroutine

© 1976 PROGRAMMA CONSULTANTS

	SBC	Subtract Memory from Accumulator with Borrow
*	SEC	Set Carry Flag
*	SED	Set Decimal Mode
*	SEI	Set Interrupt Disable Status
	STA	Store Accumulator in Memory
	STX	Store Index X in Memory
	STY	Store Index Y in Memory
*	TAX	Transfer Accumulator to Index X
*	TAY	Transfer Accumulator to Index Y
*	TSX	Transfer Stack Pointer to Index X
*	TXA	Transfer Index X to Accumulator
*	TXS	Transfer Index X to Stack Pointer
*	TYA	Transfer Index Y to Accumulator

* Instructions legal only in the implied addressing mode

© 1976 PROGRAMMA CONSULTANTS

ADDRESSING MODES

The addressing modes of the MCS6500 family can be grouped into two major categories: Indexed and Non-Indexed Addressing. Subsequent to reading this section the user should have an understanding of the following fundamentals:

1. Memory Field
2. Address Bus
3. Data Bus
4. Cycle Timing
5. Program Sequence
6. Pipelining

These concepts are explained in the publication "MCS6500 Microcomputer Family Programming Manual" by MOS Technology, Inc.

ABSOLUTE ADDRESSING

The absolute addressing mode is the most common in concept. The data following the machine code is treated as the address of a memory location containing the actual data to be processed during the instruction step. This address is stored in reverse order as low-byte then high-byte to increase processing efficiency during execution time.

For example:

```
0066  1050                    PIA    =$4C1F
0067  1050                    LATCH  =$4DE2
0068  1050                    BUFF1  =$54B0
0069  1050                    START  =$A000
0070  1050                    EXTRTN =$C800
0071  1050  2C 1F 4C                 BIT PIA
0072  1053  CD D7 C2                 CMP $C2D7
0073  1056  CE BA 54                 DEC BUFF1+10
0074  1059  4D B0 54                 EOR BUFF1
0075  105C  4C 00 A0                 JMP START
0076  105F  20 00 C8                 JSR EXTRTN
0077  1062  AD 5F D8                 LDA %1101100001011111
0078  1065  6E CF 11                 ROR $11CF
0079  1068  ED CF 54                 SBC BUFF1+$1F
0080  106B  8D E2 4D                 STA LATCH
```


© 1976 PROGRAMMA CONSULTANTS

PAGE ZERO ADDRESSING

In practice, the zero page addressing mode (identical in concept to the absolute addressing) is the most frequently used. This allows the expression of the instruction to be two bytes instead of three. The low byte of the data address is taken from memory, and the high byte is assumed to be zero. All instructions legal in absolute mode are also legal in zero page mode with the exception of the JMP and JSR instructions (see Table 1-0). The Assembler automatically generates the shortest possible code. It is good programming practice to reserve page zero (memory locations 0 to 255) for declaration of variables.

For example:

```
0083  106E                MODE   =$06
0084  106E                KEY    =$0C
0085  106E                COUNTR =$37
0086  106E                TTYBUF =$6B
0087  106E  65 37                ADC COUNTR
0088  1070  24 7A                BIT $7A
0089  1072  C4 06                CPY MODE
0090  1074  E6 38                INC COUNTR+1
0091  1076  A6 6B                LDX TTYBUF
0092  1078  46 9A                LSR $21+@171
0093  107A  05 0C                ORA KEY
0094  107C  86 AA                STX TTYBUF+$3F
```

IMMEDIATE ADDRESSING

The immediate mode of addressing is coded by the character "#" followed by a byte expression. The code inserted into memory is treated as the data to be operated upon according to the machine code.

For example:

```
0059  1046                DOLLAR =$24
0060  1046  69 03          LAB1   ADC #3
0061  1048  29 B5                AND #%10110101
0062  104A  E0 24                CPX #DOLLAR
0063  104C  A9 45                LDA #'E
0064  104E  A0 46                LDY #<LAB1
```


© 1976 PROGRAMMA CONSULTANTS

IMPLIED ADDRESSING

Twenty-five of the fifty-six machine instructions, legal only in the implied addressing mode, require no operand. Their execution may be completed with no other information than that contained in the opcode. These instructions are preceded by a * in Table 1-0.

For example:

```
0041  1033  00          BRK
0042  1034  D8          CLD
0043  1035  C8          INY
0044  1036  EA          NOP
0045  1037  68          PLA
0046  1038  60          RTS
0047  1039  8A          TXA
```

ACCUMULATOR ADDRESSING

Instructions implementing the four shift operations have, in addition to addressing modes referencing memory, a special mode which allows manipulation of the accumulator. Usage of this mode, similar to implied addressing, causes generation of a single byte of machine code.

For example:

```
0049  103A  0A          ASL A
0050  103B  4A          LSR A
0051  103C  2A          ROL A
0052  103D  6A          ROR A
```

RELATIVE ADDRESSING

There are eight conditional branch instructions available to the programmer. Normally, these immediately follow load, compare, arithmetic, and shift instructions. Branch instructions uniquely use the relative addressing mode. The branch address is a one-byte positive or negative offset, expressed in twos complement notation, from the run-time program counter. At the time the branch address calculation is made, the program counter points to the first memory location beyond the branch instruction code. Hence, the one-byte offset limits access to branch addresses within 129 bytes forward and 126 bytes backward from the beginning of the branch code. (A one-byte twos complement number is limited to the range -128 to +127 inclusive.) An error will be flagged at assembly time if the branch target lies out-of-bounds for relative addressing.

For example:

© 1976 PROGRAMMA CONSULTANTS

```
0054  103E  90 80        BCC *-126
0055  1040  F0 FE  HERE  BEQ HERE
0056  1042  30 FC        BMI *-2
0057  1044  70 7F        BVS *+129
```

INDEXED ADDRESSING

Indexed addressing (possible with index registers X or Y) facilitates certain types of table processing. The address given as the operand is treated as the base address, to which the contents of either the X or the Y register is added to arrive at the actual address of the memory location containing the data to be operated upon. All instructions implementing absolute indexed addressing with the X register also allow the same addressing in the page zero mode. Several instructions (LDX, LDY, STX, and STY) allow zero page indexed addressing with the Y register.

For example:

```
0096  107E              ARRAY  =$0B
0097  107E              NUMBUF =$50
0098  107E              TABLE  =$22C0
0099  107E  79 4F 00           ADC NUMBUF-1,Y
0100  1081  DD C0 22           CMP TABLE,X
0101  1084  D6 0B              DEC ARRAY,X
0102  1086  5D CC 22           EOR TABLE+$C,X
0103  1089  B6 0B              LDX ARRAY,Y
0104  108B  36 22              ROL >TABLE,X
0105  108D  96 8F              STX NUMBUF+$3F,Y
```

INDIRECT ADDRESSING

The concept of indirect addressing constitutes a level of complexity beyond that of absolute addressing. The operand address references not one memory location containing data, but a sequence of two memory locations containing the address, stored in low-byte - high-byte order, of the location containing the actual data to be processed. True indirect addressing is offered only with the JMP instruction; otherwise, indexed indirect addressing with the X register and indirect indexed addressing with the Y register are implemented. For indexed indirect addressing, the indexed address is computed before the indirect is taken. The order of evaluation is reversed for indirect indexed addressing. Note that normal indirect addressing takes place when the index register contains zero. The JMP indirect uses an absolute-length (two-byte) operand; others require the operand address to lie in page zero between 0 and 254 inclusive.

For example:

© 1976 PROGRAMMA CONSULTANTS

```
0107  108F                INDADR =$02
0108  108F                CURSOR =$57
0109  108F                OLDPTR =$7E
0110  108F                NEXT   =$D9
0111  108F  21 02              AND (INDADR,X)
0112  1091  D1 7E              CMP (OLDPTR),Y
0113  1093  6C D9 00           JMP (NEXT)
0114  1096  B1 57              LDA (CURSOR),Y
0115  1098  E1 02              SBC (INDADR,X)
0116  109A  81 57              STA (CURSOR,X)
```


© 1976 PROGRAMMA CONSULTANTS

4. ASSEMBLER INSTRUCTIONS

SOURCE STATEMENTS OVERVIEW

Assembler statements are requests to the assembler to perform certain operations during the assembly. Some statements, such as .BYTE and .WORD, generate data, while others, such as .PAGE and .SKIP, are effective only at assembly time. A summary of assembler statements can be found in Appendix E.

Assembler instruction source statements consist of:

Name field (usually optional)

Mnemonic operation code

Operand field (optional for some statements)

Comment field (optional)

An example of an assembler instruction source statement is:

	NAME	OP	OPERAND	COMMENT
	1-10	11-20	21-30	31-40 41-50
	1234567890	1234567890	1234567890	1234567890 1234567 8
1	NAME1	.BYT	'ABCD',0,'JOE''S'	
2				

The following is a list of assembler statements with a short explanation for each statement.

PROGRAM CONTROL STATEMENTS

.END Ends the assembly

.FILE Chains two or more files in an assembly

.OPT Options selected during an assembly

.OFS Object code directed to memory

© 1976 PROGRAMMA CONSULTANTS

SYMBOL AND DATA DEFINITION STATEMENTS

=	Defines a symbol and assigns values to it
.BYT	Defines byte oriented data in memory
.WOR	Defines address constants
.DBY	Defines word oriented (16-bits) data in memory

LISTING CONTROL STATEMENTS

.PAGE	Identifies the listing & causes page eject
.SKIP	Insert blank lines

PROGRAM CONTROL STATEMENTS

.END - END ASSEMBLY

The END statement terminates the assembly of a program. The format of the END statement is:

NAME	OPERATION	OPERAND
Blank	.END	- none -

The END statement is the last physical statement of any assembly language program. In the case where the user is using multiple chained source files, the END statement is located as the last source statement of the last chained file.

For example:

```
START   LDA  #0          ; SET A TO ZERO
        JSR  TIMEDL      ; APPLE DELAY
        JSR  PRINT       ; RING THE BELL
        JMP  START       ; DO THIS FOREVER
        .END             ; END ASSEMBLY
```


© 1976 PROGRAMMA CONSULTANTS

.FILE - CHAIN FILES

The FILE statement allows the chaining of several files of assembly language source code. The format of the FILE statement is:

NAME	OPERATION	OPERAND
Blank	.FILE	'filename'

For large programs, it is usually more convenient to divide the program into logical segments which may be easily edited. However, when the entire program is to be assembled, it is necessary to tie these files together. This function is performed with the FILE and the END assembler instructions.

Each file contains, as its last record, a FILE directive which points to the next file in the chain. The last file in the chain contains an END directive.

The ASM/65 Assembler makes its first pass (PASS #1) on the text edit buffer until it encounters either the END or FILE directives. When the FILE directive is found, the present file is saved into a temporary file called "TEMP**". The Assembler then proceeds to load the file specified by the name in the operand field of the FILE directive. Note that the 'filename' must be enclosed in apostrophes. The file loading procedure continues until the END directive is found. At this point, the file in "TEMP**" is re-loaded and PASS #2 is initiated.

For example:

```
        .PAGE   'TEST'          ; TITLE
        .OPT    NOS             ; NO SYMBOL TABLE
        TIMEDL =$FCA8           ; TIME DELAY SUB
        OUTPUT =$FDED           ; APPLE OUTPUT RTN
        *=$1000                 ; ORG AT $1000
START   LDA     #0              ; SET A TO ZERO
        JSR     TIMEDL          ; APPLE DELAY
        JSR     PRINT           ; RING THE BELL
        JMP     START           ; DO THIS FOREVER
        .FILE   'TEST2'         ; CHAIN NEXT FILE
```


© 1976 PROGRAMMA CONSULTANTS

```
        .PAGE   'TEST2'         ; TITLE SECOND FILE
PRINT   LDA     #%00000111      ; ASCII BELL ($07)
        ORA     #$80            ; APPLE LIKES THIS
        JSR     OUTPUT          ; RING THE BELL
        RTS                     ; RETURN FROM CALL
        .END                    ; END OF ASSEMBLY
```

In the above example, the editor would be used to create the first file, and it would be saved under the name 'TEST1'. The editor is once again used to create the second file and it is saved under the name 'TEST2'.

.OPT - OPTIONS

The OPT statement specifies user selectable options that will be in effect during the assembly of the source program. The format of the OPT statement is:

NAME	OPERATION	OPERAND
Blank	.OPT	OPT1,OPT2,OPT3,...

There are a possible of six (6) OPT operands that can be supplied to the OPT directive. In addition, each of the six operands can be negated in order to override the ASM/65 default options. The user may code only the first three (3) significant characters of each option if he chooses.

The six possible operands to the OPT directive are:

```
        .OPT    LIST,GENERATE,ERRORS,SYMBOL,MEMORY,OBJECT
```

and these are eliminated or negated by:

```
        .OPT    NOLIST,NOGENERATE,NOERRORS,NOSYMBOL,NOMEMORY,NOOBJECT
```

Since only the first three characters of each option are scanned, each of the previous could be written as:

```
        .OPT    LIS,GEN,ERR,SYM,MEM,OBJ

        .OPT    NOL,NOG,NOE,NOS,NOM
```


© 1976 PROGRAMMA CONSULTANTS

Listed below is an explanation for each of the operands of the OPT directive.

LIST [NOLIST]

Controls generation of the program listing, which contains assembled source input, generated object code, errors and warnings.

GENERATE [NOGENERATE]

Controls the printing of object code for ASCII strings in the BYTE directive. Only code for the first two characters is listed in NOG is specified; otherwise, the whole literal will be expanded.

ERRORS [NOERRORS]

Controls the listing of only those lines that contain errors in the source listing with the respective error message generated. Fatal assembler table overflows are also listed.

SYMBOL [NOSYMBOL]

Controls the listing of the symbol table at the end of the assembly. Also, it controls retaining the symbol table in memory for subsequent use in disassembling instructions during debugging traces.

MEMORY [NOMEMORY]

Controls the storing of the object code generated for the program in RAM memory. Care should be used so as not to overlay portions of the Assembler.

OBJECT 'filename'

Controls the generation of the object code to a sequential file on disk called 'filename'. This option cannot be used with the NOM option. The object code file is built starting at location $800 for the length of the program; when the assembly is completed, the buffer is saved to 'filename' on disk. Note that there exists no NOO option.

For example:

```
.OPT    LIS,NOS,NOG,ERR
.OPT    OBJ 'TEST1'
```


© 1976 PROGRAMMA CONSULTANTS

The ASM/65 Editor Assembler default options are:
GEN,NOE,SYM,MEM

.OFS - OFFSET

The OFS statement is used to route object code generated by the ASM/65 Editor Assembler to a positive page offset boundary. Typically, if the user wants to generate object code using the MEM option of the OPT statement, the Assembler begins placing code in memory beginning at the location specified by the setting of the location counter. In those cases where the code is required to run, but the physical memory space is being used by the Assembler or the text edit buffer, the user uses the OFS statement to temporarily route the code. After the assembly is completed, the offset object code is moved to the proper location and executed. The format of the OFS statement is:

NAME	OPERATION	OPERAND
Blank	.OFS=	Decimal Integer

The best method of understanding the OFS statement is through an example. Suppose that the user wanted to assemble a source program, which is designed to run at location $7DØØ. The location counter in the source program would be set to origin at $7DØØ as follows:

```
*=$7DØØ      ; ORG AT $7DØØ
```

Since the Assembler presently occupies the area that the object code is being generated to, the user must use the OFS statement to route the object code to a different location temporarily. Let's assume that we would like to place the object code temporarily at location $Ø8ØØ. Now, the user has to compute the positive page number where the object code is to be placed. The formula for computing the positive page offset is as follows:

> Subtract the HEX value of the address that the program is ORGinated to from HEX FFFF. Add one (1) to the result of the subtraction. Add to the new number the HEX address that the program is being OFSet to. Convert this new number to decimal. Divide the number by 256, and the answer is the positive page offset.

Example:

```
$FFFF  -  $7DØØ   =   $82FF      Subtract ORG Address
$82FF  +  $ØØØ1   =   $83ØØ      Add 1
$83ØØ  +  $Ø8ØØ   =   $8BØØ      Add Ofset address
$8BØØ  to Base 10 =   35584       Convert to Decimal
35584  /    256   =   139         Positive Page Offset
```


© 1976 PROGRAMMA CONSULTANTS

To specify the page offset in our particular example, the user would code the following:

```
.OFS=8          ; OFFSET OBJECT CODE
```

Please note that caution should be used with the OFS command as the user may accidentally specify the wrong offset and overwrite the Assembler. Also, please do not forget the "=" that follows the OFS, since this version does not check syntax.

= - SET LOCATION COUNTER

The = statement alters the setting of the location counter. The format of the = statement is:

NAME	OPERATION	OPERAND
blank	*=	Expression

Since the location counter points to a storage location that is to receive the next line of assembled code, altering its setting permits a programmer to return to a previous location in his program and change its contents. In this way, an area can be redefined during assembly, changing data definitions to meet various requirements in the program. An example of the = statement is:

```
*=$0800
```

The = statement can also be used in direct reference to the location counter. The format of the operand is *-n or *+n. The asterisk specifies the present location counter, and n specifies the number of bytes. The following statement decreases the present location counter by 60:

```
*=*-60
```

Note that any valid base can be used in describing the integer number specified in the operand field of the = statement.

For example:

```
*=1000
*=$800
*=@756
```


© 1976 PROGRAMMA CONSULTANTS

= - EQUATE

The = statement also assigns values to a symbol. The other format of the = statement is:

NAME	OPERATION	OPERAND
Symbol	=	Expression

The expression in the operand field must be absolute; all symbols in the expression must be previously defined. The value attribute of the symbol is the value of the expression. When a newly defined symbol is referenced in later statements, it has the attributes assigned by the = statement.

Terms that use the location counter reference can be used in the expression.

The = statement can equate symbols to immediate data and other arbitrary values. Note that all symbols in the expression must be previously defined.

For example:

```
      NAME      OP      OPERAND          COMMENT
      1-10      11-20   21-30            31-40            41-50
1                       *=$1800          ; SET LOCATION COUNTER
2     TABLE2            =$C800           ; DEFINE SYMBOLS
3     WRDPTR            =$2A
4     NUMPTR            =WRDPTR+2
5     CURSOR            =*+5
6                       *=800            ; SET LOC CTR
7     EOT               *=*+1            ; RESERVE
8     EOTADR            *=*+2            ; MEMORY
9     CR                =$0D             ; CONSTANT
10
```


© 1976 PROGRAMMA CONSULTANTS

.BYT - BYTE DEFINED DATA

The BYTE statement initializes and defines byte data in storage. The format of the BYTE statement is:

NAME	OPERATION	OPERAND
Symbol/ blank	.BYT	One or more operands separated by commas

The value attribute of the symbol in the name field is the address of the leftmost byte of the operand. Multiple arguments, separated by commas, may be specified in a single BYTE statement to initialize consecutive memory locations.

ASCII strings or expressions evaluating to an 8-bit value are legal. ASCII strings in BYTE statements must not generate more that 20 characters.

Two quotes within an ASCII string are used to insert a single quote into storage. ASCII strings must be enclosed within quotes.

For example:

```
0141   1800   41 42          ASCII  .BYT 'ABCD','EFGH','JOE''S'
0141   1802   43 44
0141   1804   45 46
0141   1806   47 48
0141   1808   4A 4F
0141   180A   45 27 53
0142   180D   00                    .BYT <ASCII,>ASCII+2-%1,<*+>*,2
0142   180E   19
0142   180F   27
0142   1810   02
```

Omitted operands, signified by a comma without a data value, indicate a zero byte. The last data value in a string of multiple operands must be a specified data value, not an omitted operand.

© 1976 PROGRAMMA CONSULTANTS

.WOR - WORD DEFINED DATA

The WORD statement defines and initializes word data in storage. The format of the WORD statement is:

NAME	OPERATION	OPERAND
Symbol/ blank	.WOR	One or more operands separated by commas

The WORD statement is very useful in constructing jump tables and initializing pointers. An operand expression is evaluated as a two-byte address and is stored in low-byte, high-byte sequence; the order which the microprocessor fetches addresses from memory.

The value attribute of the symbol in the name field is the address of the leftmost byte of the operand. Multiple arguments, separated by commas, may be specified in a single WORD statement to initialize consecutive pairs of storage locations.

For example:

```
0144  1811  04 C8     JMPTBL  .WOR   $C804,$E4B9,$F77A
0144  1813  B9 E4
0144  1815  7A F7
0145  1817  02 00             .WOR   $2,<JMPTBL,>JMPTBL,*,@6371
0145  1819  11 00
0145  181B  18 00
0145  181D  1D 18
0145  181F  F9 0C
```

.DBY - WORD DEFINED DATA

The DBYTE statement is used to generate a 16-bit expression value in normal high-byte, low-byte order. The syntax rules for the use of the DBYTE statement are identical to the WORD statement. The format of the DBYTE statement is:

NAME	OPERATION	OPERAND
Symbol/ blank	.DBY	One or more operands separated by commas

© 1976 PROGRAMMA CONSULTANTS

The value attribute of the symbol in the name field is the address of the leftmost byte of the operand. Multiple arguments, separated by commas, may be specified in a single DBYTE statement to initialize consecutive pairs of storage locations.

For example:

```
0148  1821  C8 04      DATA    .DBY   $C804,$E4B9,$F77A
0148  1823  E4 B9
0148  1825  F7 7A
0149  1827  00 0E              .DBY   $E,<DATA,>DATA,*,%0101101111101
0149  1829  00 21
0149  182B  00 18
0149  182D  18 2D
0149  182F  05 BD
```

LISTING CONTROL STATEMENTS

.PAGE- IDENTIFY LISTING/START NEW PAGE

The PAGE statement is used both to cause a page eject (top-of-form) and to generate or alter the title printed at the top of the new page. Both of these features apply only to the assembler source listing. The format of the PAGE statement is:

NAME	OPERATION	OPERAND
blank	.PAGE	ASCII String or blank

A title may be specified as an ASCII string in the operand field enclosed within quotation marks. The title may be cleared by coding a string of one or more blanks.

Absence of an operand will cause the title printed on the previous page, if any, to be repeated at the top of the new page.

Note that this command is not printed in the source listing.

For example:

```
.PAGE   'MAIN LOGIC SECTION'
.PAGE
.PAGE   ' '
```


© 1976 PROGRAMMA CONSULTANTS

.SKIP - INSERT BLANK LINES

The SKIP statement inserts one or more blank lines in a listing. The format of the SKIP statement is:

NAME	OPERATION	OPERAND
blank	.SKIP	Absolute expression

The Assembler evaluates the expression in the operand field to determine the number of lines to leave blank. Symbols in the expression must be previously defined. If the number of lines to be left blank exceeds the number left on the current page, the SKIP statement functions as a PAGE statement, and the next line is printed on the top of the next page.

If no operand is given, one line will be skipped.

Note that this command is not printed in the source listing.

For example:

```
.SKIP
.SKIP  6
.SKIP  LINES+3
```


© 1976 PROGRAMMA CONSULTANTS

5. USING THE ASSEMBLER

LOADING THE ASSEMBLER

The ASM/65 EDITOR ASSEMBLER is supplied on an Apple II compatible 5-1/4" diskette. The following files are contained on the diskette:

```
ASM\65 [DOS 3.2]
HEADER
ASM/65 APPLE ][
TEST PART1
TEST PART2
TEMP**
TEST OBJECT
```

The ASM\65 [DOS 3.2] file contains the "HELLO" program for the Apple II, and it in turn uses the HEADER file to display the "PROGRAMMA SOFTWARE" logo page. The ASM/65 APPLE][file contains the actual code of the ASM/65 EDITOR ASSEMBLER. The TEST PART1 and TEST PART2 files are a sample program that is distributed with the system. The TEMP** file is a required file that is used whenever the FILE statement is used within an assembler language program. The TEST OBJECT file contains the object code generated by the assembly of the supplied sample program.

To run the ASM/65 EDITOR ASSEMBLER, the following procedure is used:

1. RESET the Apple II Computer
   The system monitor should show an "*" on the CRT.

2. Place the ASM/65 EDITOR ASSEMBLER diskette in the DISK II drive.

3. Enter on the keyboard the SLOT NUMBER of the DISK II drive followed by a "Control P" and "RETURN" sequence.

4. Depress the "Space Bar" when the "PROGRAMMA SOFTWARE" logo page is displayed.
   The code for the Assembler will then be loaded.

5. The ASM/65 EDITOR ASSEMBLER should prompt the user with the following text:

   COMMAND?

6. Enter a "RETURN" to obtain the Function Menu.

© 1976 PROGRAMMA CONSULTANTS

FUNCTION MENU

At any point in time that the Assembler is prompting the user with the text "COMMAND?", the user may enter the "RETURN" key to obtain a listing of the Function Menu. This menu contains the commands that are valid in the "Command Processor Mode" of the ASM/65 EDITOR ASSEMBLER. Below is a list of the valid commands and the use of each:

A	Assemble source code in the text edit buffer
C	Catalogue the disk
E	Edit a new program through ApplePIE Text Editor
L	Load a source code file from disk
P	Print the contents of the text edit buffer
Q	Quit, return to the Apple II monitor
R	Re-edit the program through ApplePIE Text Editor
S	Save a source code file to disk
RETURN	Display Function Menu

A - ASSEMBLE

Before a source program can be assembled, there must exist assembler source statements in the text edit buffer. The user may edit a new program by using the "E" command and then exiting the ApplePIE Text Editor back to the Assembler's command processor mode. On the other hand, an old source program can be loaded into the text edit buffer by using the "L" command. Once there exists text in the text edit buffer, the user may issue the "A" command to initiate an assembly.

For example: (Entries underscored signify input by the user)

```
COMMAND? L
.
.
CATALOGUE DISPLAY OF FILES
.
.
LOAD FROM DISK --->
FILE NAME? TEMP PART1
COMMAND? A
```


© 1976 PROGRAMMA CONSULTANTS

```
SCREEN OR PRINTER (S/P)? S
PASS1
PASS2
.
.
SOURCE LISTING
.
.
COMMAND?
```

When an assembly is performed by using the "A" command, the system will prompt the user with the following text:

```
SCREEN OR PRINTER (S/P)?
```

This prompt is used to allow the user to select the output device that the source listing will be routed to. Should the user select the SCREEN option ("S"), the source listing will be formatted to fit the CRT device. Here the keyboard's space bar may be used to stop and start the listing during an assembly for ease of reading.

In the case where the PRINTER option ("P") is selected, the user will be asked the SLOT NUMBER where the printer is interfaced to. The user should enter the slot number followed by a RETURN. Note that the Assembler's print driver routines are designed to operate with an Apple intelligent controller only.

C - CATALOG

The "C" command is used to obtain a listing of the diskette's catalog. This command allows the user to view the names of the files that are contained in a diskette, without leaving the Assembler.

For example:

```
COMMAND? C
.
.
*I 002  ASM\65 [DOS 3.2]
*B 007  HEADER
*B 046  ASM/65 APPLE ][
 B 006  TEST PART1
 B 002  TEST PART2
 B 006  TEMP**
 B 002  TEST OBJECT
```


© 1976 PROGRAMMA CONSULTANTS

E - EDIT

The "E" command is used to gain a cold-start entry into the ApplePIE Text Editor System. Any text that may have been in the text edit buffer is lost once the "E" command is entered. The user is advised to read the User's Reference Manual on the ApplePIE Text Editor for further information and documentation on its use.

L - LOAD

Source text can be loaded into the text edit buffer from diskette by using the "L" command. This command will cause a catalog display of the diskette to occur automatically. The "L" command will also prompt the user for the name of the diskette file to be loaded.

For example:

```
COMMAND? L
.
.
CATALOGUE DISPLAY OF FILES
.
.
LOAD FROM DISK -->
FILE NAME? TEMP PART1
COMMAND?
```

P - PRINT

The "P" command is used to print the contents of the text edit buffer to a printer. This command will prompt the user with the PORT# that the printer is connected to. The user should enter the port number in responce to the prompt.

Q - QUIT

The "Q" command is used to quit the ASM/65 EDITOR ASSEMBLER and to return to the Apple II monitor. It will be necessary to re-boot the DOS 3.2 System from another diskette if the user wishes to run under control of the disk operating system. The ASM/65 EDITOR ASSEMBLER System modifies several vectors within the DOS 3.2 System, so that any disk I/O errors are vectored to the Assembler rather than BASIC. Therefore, if the user plans to use the DOS 3.2 for further service after using the Assembler, remember to re-boot from another diskette.

© 1976 PROGRAMMA CONSULTANTS

R - RE-EDIT

The "R" command is used to re-edit source text in the text edit buffer through the ApplePIE Text Editor System. The user is advised to read the User's Reference Manual on the ApplePIE Text Editor for detailed information and documentation on its use.

S - SAVE

Source text that resides in the text edit buffer can be saved to diskette by using the "S" command. This command will cause a catalog display of the diskette to occur automatically. The "S" command will also prompt the user for the name of the file.

For example:

```
COMMAND? S
.
.
CATALOGUE DISPLAY OF FILES
.
.
SAVE TO DISK -->
FILE NAME? COBOL.ASM
COMMAND?
```

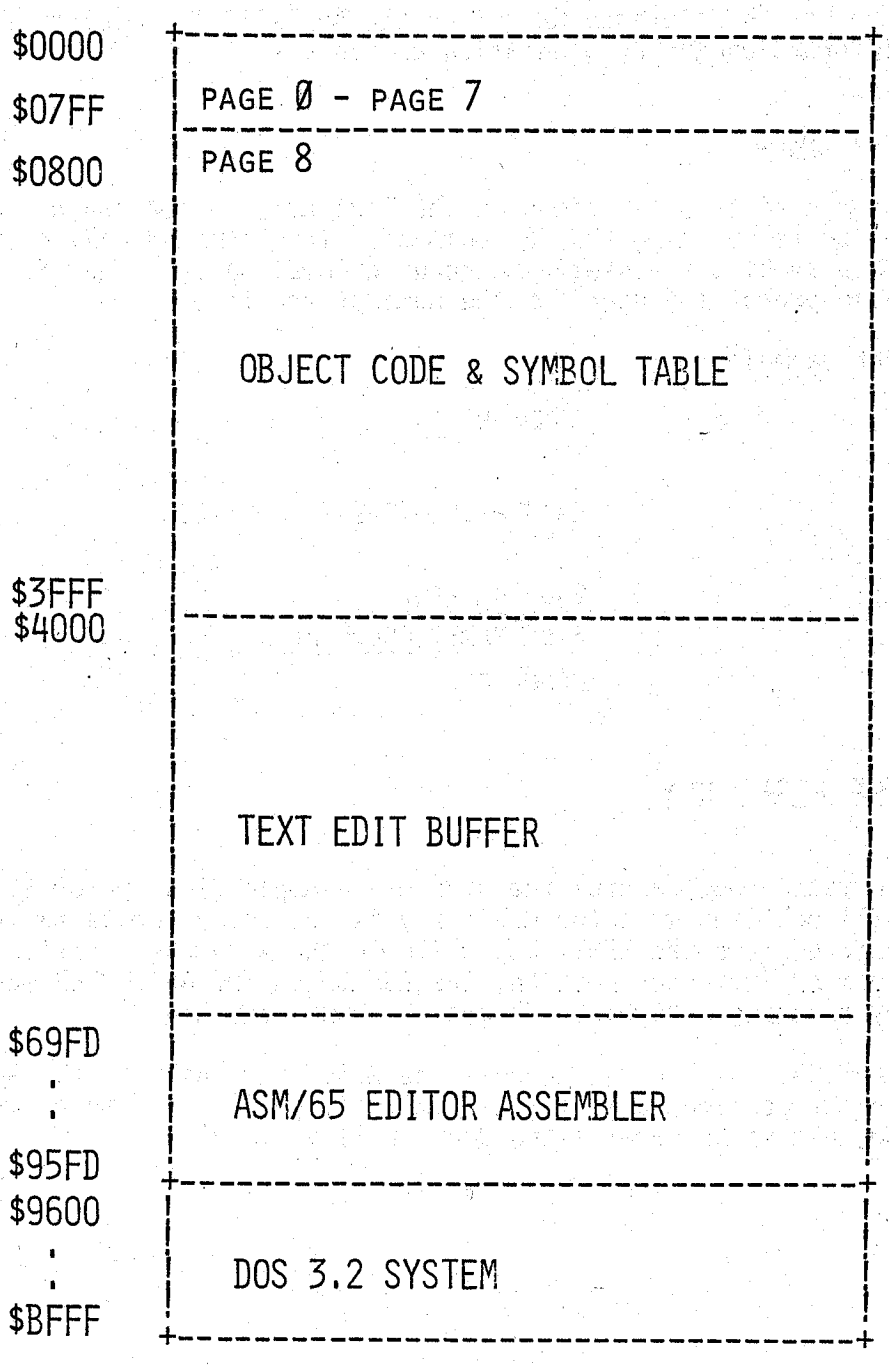
THE RESET KEY

No matter how careful the user of an Apple II Computer System is, there will be times when the RESET key is depressed accidentally. Should the user depress the RESET key while in the Assembler portion or in the Command Processor portion, the ASM/65 EDITOR ASSEMBLER may be re-entered by entering a "CONTROL Y" and "RETURN" sequence.

CAUTION! Should you depress the RESET key while in the ApplePIE Text Editor portion, all of your text will be lost and there is no recovery. So, please be careful and don't pull a dummy!

© 1976 PROGRAMMA CONSULTANTS

6. MEMORY MAP



© 1976 PROGRAMMA CONSULTANTS

A. ASCII REPRESENTATION

The letters in parenthesis under the heading KEY below have the following meanings: c = Control, s = Shift. These must be held down while striking the other key. In addition, most terminals provide separate keys for commonly used control characters, e.g. Line Feed, Carriage Return, etc.

DEC	HEX	SYMBOL	KEY	MEANING
000	000	NUL	P (c s)	NULL
001	001	SOH	A (c)	Start of Heading
002	002	STX	B (c)	Start of Text
003	003	ETX	C (c)	End of Text
004	004	EOT	D (c)	End of Transmission
005	005	ENQ	E (c)	Enquiry
006	006	ACK	F (c)	Acknowledge
007	007	BEL	G (c)	BELL
008	008	BS	H (c)	Backspace
009	009	HT	I (c)	Horizontal Tab
010	00A	LF	J (c)	Line Feed (also New Line)
011	00B	VT	K (c)	Vertical Tab
012	00C	FF	L (c)	Form Feed (also Top of Form)
013	00D	CR	M (c)	Carriage Return (also Return)
014	00E	SO	N (c)	Shift Out
015	00F	SI	O (c)	Shift In
016	010	DLE	P (c)	Data Link Escape
017	011	DC1	Q (c)	Device Control 1
018	012	DC2	R (c)	Device Control 2

© 1976 PROGRAMMA CONSULTANTS

DEC	HEX	SYMBOL	KEY	MEANING
019	013	DC3	S (c)	Device Control 3
020	014	DC4	T (c)	Device Control 4
021	015	NAK	U (c)	Negative Acknowledge
022	016	SYN	V (c)	Synchronous Idle
023	017	ETB	W (c)	End of Transmission Block
024	018	CAN	X (c)	Cancel
025	019	EM	Y (c)	End of Medium
026	01A	SUB	Z (c)	Substitute
027	01B	ESC	K (c s)	Escape
028	01C	FS	L (c s)	File Separator
029	01D	GS	M (c s)	Group Separator
030	01E	RS	N (c s)	Record Separator
031	01F	US	O (c s)	Unit Separator
032	020	SP	Space	Space (also Blank)
033	021	!	!	Exclamation Point
034	022	"	"	Quotation Mark (Double)
035	023	#	#	Number Sign
036	024	$	$	Dollar Sign
037	025	%	%	Percent Sign
038	026	&	&	Ampersand
039	027	'	'	Apostrophe (Single)
040	028	(	(	Opening Parenthesis
041	029	)	)	Closing Parenthesis
042	02A	*	*	Asterisk

© 1976 PROGRAMMA CONSULTANTS

DEC	HEX	SYMBOL	KEY	MEANING
043	02B	+	+	Plus
044	02C	,	,	Comma
045	02D	-	-	Hyphen (Minus)
046	02E	.	.	Period (Decimal Point)
047	02F	/	/	Slant
048	030	0	0	0 (Zero)
049	031	1	1	1 (One)
050	032	2	2	2
051	033	3	3	3
052	034	4	4	4
053	035	5	5	5
054	036	6	6	6
055	037	7	7	7
056	038	8	8	8
057	039	9	9	9
058	03A	:	:	Colon
059	03B	;	;	Semicolon
060	03C	<	<	Less Than
061	03D	=	=	Equals
062	03E	>	>	Greater Than
063	03F	?	?	Question Mark
064	040	@	@	Commercial At
065	041	A	A	A
066	042	B	B	B

© 1976 PROGRAMMA CONSULTANTS

DEC	HEX	SYMBOL	KEY	MEANING
067	043	C	C	C
068	044	D	D	D
069	045	E	E	E
070	046	F	F	F
071	047	G	G	G
072	048	H	H	H
073	049	I	I	I
074	04A	J	J	J
075	04B	K	K	K
076	04C	L	L	L
077	04D	M	M	M
078	04E	N	N	N
079	04F	O	O	O
080	050	P	P	P
081	051	Q	Q	Q
082	052	R	R	R
083	053	S	S	S
084	054	T	T	T
085	055	U	U	U
086	056	V	V	V
087	057	W	W	W
088	058	X	X	X
089	059	Y	Y	Y
090	05A	Z	Z	Z

© 1976 PROGRAMMA CONSULTANTS

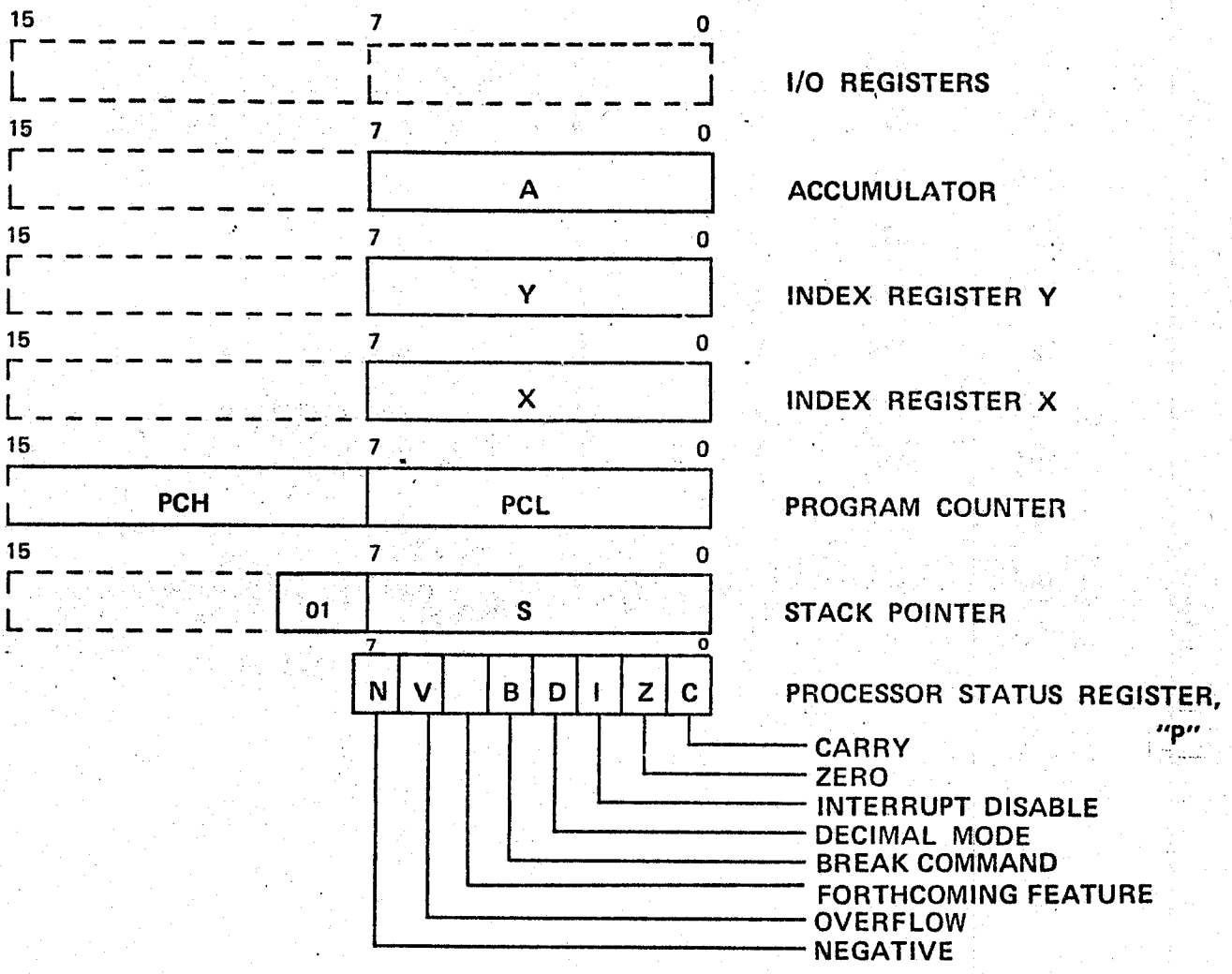
DEC	HEX	SYMBOL	KEY	MEANING
091	05B	[	[	Opening Bracket
092	05C	\	\	Reverse Slant
093	05D	]	]	Closing Bracket
094	05E	^	^	Circumflex (also Up Arrow)
095	05F	_	_	Underline (also Backarrow)
096	060	¢ or `	¢ or `	Cent Sign (also Grave Accent)
097	061	a	a	a
098	062	b	b	b
099	063	c	c	c
100	064	d	d	d
101	065	e	e	e
102	066	f	f	f
103	067	g	g	g
104	068	h	h	h
105	069	i	i	i
106	06A	j	j	j
107	06B	k	k	k
108	06C	l	l	l
109	06D	m	m	m
110	06E	n	n	n
111	06F	o	o	o
112	070	p	p	p
113	071	q	q	q
114	072	r	r	r

© 1976 PROGRAMMA CONSULTANTS

DEC	HEX	SYMBOL	KEY	MEANING
115	073	s	s	s
116	074	t	t	t
117	075	u	u	u
118	076	v	v	v
119	077	w	w	w
120	078	x	x	x
121	079	y	y	y
122	07A	z	z	z
123	07B	{	{	Opening Brace
124	07C	|	|	Vertical Line
125	07D	}	}	Closing Brace
126	07E	~	~	Overline (also Tilde)
127	07F	DEL	DELETE	Delete (also Rubout)

© 1976 PROGRAMMA CONSULTANTS

B. PROGRAMMING MODEL MCS650X



* Solid line indicates currently available features
Dashed line indicates forthcoming members of family

© 1976 PROGRAMMA CONSULTANTS

C. ALPHABETICAL LIST OF MNEMONICS

MNEMONIC	DESCRIPTION
ADC	Add Memory to Accumulator with Carry
AND	"AND" Memory with Accumulator
ASL	Shift Left One Bit (Memory or Accumulator)
BCC	Branch on Carry Clear
BCS	Branch on Carry Set
BEQ	Branch on Result Zero
BIT	Test Bits in Memory with Accumulator
BMI	Branch on Result Minus
BNE	Branch on Result not Zero
BPL	Branch on Result Plus
BRK	Force Break
BVC	Branch on Overflow Clear
BVS	Branch on Overflow Set
CLC	Clear Carry Flag
CLD	Clear Decimal Mode
CLI	Clear Interrupt Disable Bit
CLV	Clear Overflow Flag
CMP	Compare Memory and Accumulator
CPX	Compare Memory and Index X
CPY	Compare Memory and Index Y

© 1976 PROGRAMMA CONSULTANTS

MNEMONIC	DESCRIPTION
DEC	Decrement Memory By One
DEX	Decrement Index X by One
DEY	Decrement Index Y by One
EOR	"Exclusive-OR" Memory with Accumulator
INC	Increment Memory by One
INX	Increment Index X by One
INY	Increment Index Y by One
JMP	Jump to New Location
JSR	Jump to New Location Saving Return Address
LDA	Load Accumulator with Memory
LDX	Load Index X with Memory
LDY	Load Index Y with Memory
LSR	Shift Right One Bit (Memory or Accumulator)
NOP	No Operation
ORA	"OR" Memory with Accumulator
PHA	Push Accumulator on Stack
PHP	Push Processor Status on Stack
PLA	Pull Accumulator from Stack
PLP	Pull Processor Status from Stack

© 1976 PROGRAMMA CONSULTANTS

MNEMONIC	DESCRIPTION
ROL	Rotate One Bit Left (Memory or Accumulator)
ROR	Rotate One Bit Right (Memory or Accumulator)
RTI	Return from Interrupt
RTS	Return from Subroutine
SBC	Subtract Memory from Accumulator with Borrow
SEC	Set Carry Flag
SED	Set Decimal Mode
SEI	Set Interrupt Disable Status
STA	Store Accumulator in Memory
STX	Store Index X in Memory
STY	Store Index Y in Memory
TAX	Transfer Accumulator to Index X
TAY	Transfer Accumulator to Index Y
TSX	Transfer Stack Pointer to Index X
TXA	Transfer Index X to Accumulator
TXS	Transfer Index X to Stack Pointer
TYA	Transfer Index Y to Accumulator

© 1976 PROGRAMMA CONSULTANTS

D. HEX CODE TO MNEMONIC CODE

00	BRK	17	RESERVED
01	ORA - (Indirect,X)	18	CLC
02	RESERVED	19	ORA - Absolute,Y
03	RESERVED	1A	RESERVED
04	RESERVED	1B	RESERVED
05	ORA - Zero Page	1C	RESERVED
06	ASL - Zero Page	1D	ORA - Absolute,X
07	RESERVED	1E	ASL - Absolute,X
08	PHP	1F	RESERVED
09	ORA - Immediate	20	JSR
0A	ASL - Accumulator	21	AND - (Indirect,X)
0B	RESERVED	22	RESERVED
0C	RESERVED	23	RESERVED
0D	ORA - Absolute	24	BIT - Zero Page
0E	ASL - Absolute	25	AND - Zero Page
0F	RESERVED	26	ROL - Zero Page
10	BPL	27	RESERVED
11	ORA - (Indirect),Y	28	PLP
12	RESERVED	29	AND - Immediate
13	RESERVED	2A	ROL - Accumulator
14	RESERVED	2B	RESERVED
15	ORA - Zero Page,X	2C	BIT - Absolute
16	ASL - Zero Page,X	2D	AND - Absolute

© 1976 PROGRAMMA CONSULTANTS

2E	ROL - Absolute	47	RESERVED
2F	RESERVED	48	PHA
30	BMI	49	EOR - Immediate
31	AND - (Indirect),Y	4A	LSR - Accumulator
32	RESERVED	4B	RESERVED
33	RESERVED	4C	JMP - Absolute
34	RESERVED	4D	EOR - Absolute
35	AND - Zero Page,X	4E	LSR - Absolute
36	ROL - Zero Page,X	4F	RESERVED
37	RESERVED	50	BVC
38	SEC	51	EOR - (Indirect),Y
39	AND - Absolute,Y	52	RESERVED
3A	RESERVED	53	RESERVED
3B	RESERVED	54	RESERVED
3C	RESERVED	55	EOR - Zero Page,X
3D	AND - Absolute,X	56	LSR - Zero Page,X
3E	ROL - Absolute,X	57	RESERVED
3F	RESERVED	58	CLI
40	RTI	59	EOR - Absolute,Y
41	EOR - (Indirect,X)	5A	RESERVED
42	RESERVED	5B	RESERVED
43	RESERVED	5C	RESERVED
44	RESERVED	5D	EOR - Absolute,X
45	EOR - Zero Page	5E	LSR - Absolute,X
46	LSR - Zero Page	5F	RESERVED

© 1976 PROGRAMMA CONSULTANTS

60 RTS
61 ADC - (Indirect,X)
62 RESERVED
63 RESERVED
64 RESERVED
65 ADC - Zero Page
66 ROR - Zero Page
67 RESERVED
68 PLA
69 ADC - Immediate
6A ROR - Accumulator
6B RESERVED
6C JMP - Indirect
6D ADC - Absolute
6E ROR - Absolute
6F RESERVED
70 BVS
71 ADC - (Indirect),Y
72 RESERVED
73 RESERVED
74 RESERVED
75 ADC - Zero Page,X
76 ROR - Zero Page,X
77 RESERVED
78 SEI
79 ADC - Absolute,Y
7A RESERVED
7B RESERVED
7C RESERVED
7D ADC - Absolute,X
7E ROR - Absolute,X
7F RESERVED
80 RESERVED
81 STA - (Indirect,X)
82 RESERVED
83 RESERVED
84 STY - Zero Page
85 STA - Zero Page
86 STX - Zero Page
87 RESERVED
88 DEY
89 RESERVED
8A TXA
8B RESERVED
8C STY - Absolute
8D STA - Absolute
8E STX - Absolute
8F RESERVED
90 BCC
91 STA - (Indirect),Y

© 1976 PROGRAMMA CONSULTANTS

92 RESERVED
93 RESERVED
94 STY - Zero Page,X
95 STA - Zero Page,X
96 STX - Zero Page,Y
97 RESERVED
98 TYA
99 STA - Absolute,Y
9A TXS
9B RESERVED
9C RESERVED
9D STA - Absolute,X
9E RESERVED
9F RESERVED
A0 LDY - Immediate
A1 LDA - (Indirect,X)
A2 LDX - Immediate
A3 RESERVED
A4 LDY - Zero Page
A5 LDA - Zero Page
A6 LDX - Zero Page
A7 RESERVED
A8 TAY
A9 LDA - Immediate
AA TAX
AB RESERVED
AC LDY - Absolute
AD LDA - Absolute
AE LDX - Absolute
AF RESERVED
B0 BCS
B1 LDA - (Indirect),Y
B2 RESERVED
B3 RESERVED
B4 LDY - Zero Page,X
B5 LDA - Zero Page,X
B6 LDX - Zero Page,Y
B7 RESERVED
B8 CLV
B9 LDA - Absolute,Y
BA TSX
BB RESERVED
BC LDY - Absolute,X
BD LDA - Absolute,X
BE LDX - Absolute,Y
BF RESERVED
C0 CPY - Immediate
C1 CMP - (Indirect,X)
C2 RESERVED
C3 RESERVED

© 1976 PROGRAMMA CONSULTANTS

C4 CPY - Zero Page
C5 CMP - Zero Page
C6 DEC - Zero Page
C7 RESERVED
C8 INY
C9 CMP - Immediate
CA DEX
CB RESERVED
CC CPY - Absolute
CD CMP - Absolute
CE DEC - Absolute
CF RESERVED
D0 BNE
D1 CMP - (Indirect),Y
D2 RESERVED
D3 RESERVED
D4 RESERVED
D5 CMP - Zero Page,X
D6 DEC - Zero Page,X
D7 RESERVED
D8 CLD
D9 CMP - Absolute,Y
DA RESERVED
DB RESERVED
DC RESERVED
DD CMP - Absolute,X
DE DEC - Absolute,X
DF RESERVED
E0 CPX - Immediate
E1 SBC - (Indirect,X)
E2 RESERVED
E3 RESERVED
E4 CPX - Zero Page
E5 SBC - Zero Page
E6 INC - Zero Page
E7 RESERVED
E8 INX
E9 SBC - Immediate
EA NOP
EB RESERVED
EC CPX - Absolute
ED SBC - Absolute
EE INC - Absolute
EF RESERVED
F0 BEQ
F1 SBC - (Indirect),Y
F2 RESERVED
F3 RESERVED
F4 RESERVED
F5 SBC - Zero Page,X

© 1976 PROGRAMMA CONSULTANTS

F6 INC - Zero Page,X

F7 RESERVED

F8 SED

F9 SBC - Absolute,Y

FA RESERVED

FB RESERVED

FC RESERVED

FD SBC - Absolute,X

FE INC - Absolute,X

FF RESERVED

© 1976 PROGRAMMA CONSULTANTS

E. SUMMARY OF ASSEMBLER STATEMENTS

PROGRAM CONTROL STATEMENTS

.END	END ASSEMBLY
.FILE	CHAIN SOURCE FILES
.OPT	SELECT ASSEMBLY OPTIONS
.OFS	OFFSET OBJECT CODE

SYMBOL AND DATA DEFINITION STATEMENTS

*=	SET LOCATION COUNTER
=	EQUATE
.BYT	DEFINE BYTE ORIENTED DATA
.WOR	DEFINE WORD ORIENTED ADDRESS DATA
.DBY	DEFINE WORD ORIENTED DATA

LISTING CONTROL STATEMENTS

.PAGE	IDENTIFY LISTING/START NEW PAGE
.SKIP	INSERT BLANK LINES

© 1976 PROGRAMMA CONSULTANTS

F. ASSEMBLER ERROR MESSAGES

ASSEMBLER SOURCE ERROR DIAGNOSTIC MESSAGES

Each assembler error message is printed in the following format:

*** ERROR # XX <AAAA>

where:

XX is the Error Code

AAAA is the line number of a previous error

The error code is a two-digit number designating which assembly error occurred. The line number of the previous error is shown to aid the user in debugging a program.

The error messages and their explanations follow.

01 UNDEFINED SYMBOL

The Assembler has found a symbol in an operand expression which is nowhere defined in the source code. This error will also occur if a reserved name is referenced as a symbol in an expression.

02 LABEL PREVIOUSLY DEFINED

The first field on the line has been found already defined with a value in the symbol table.

03 ILLEGAL OR MISSING OPCODE

The Assembler has found a line containing a label, followed by an expression, which it tried to interpret as an instruction.

04 ADDRESS NOT VALID

An address referenced in an instruction or in one of the assembler directives is invalid.

© 1976 PROGRAMMA CONSULTANTS

05 ACCUMULATOR MODE NOT ALLOWED

The instruction in the statement is one which does not allow reference to the accumulator.

06 FORWARD REFERENCE TO PAGE ZERO

An operand expression for an instruction mnemonic has been found which contains a forward reference.

07 RAN OFF END OF LINE

This message occurs when the Assembler is looking for a needed field and runs off the end of the line image before the field is found.

08 LABEL DOESN'T BEGIN WITH ALPHABETIC CHARACTER

The first character of the first non-blank field begins with a numeric character, violating the rules of symbol construction.

09 LABEL GREATER THAN SIX CHARACTERS

The first field on the line is a string containing more than six characters, violating the rules of symbol construction.

10 LABEL OR OPCODE CONTAINS NON-ALPHANUMERIC CHARACTER

The label or opcode field on a line contains a character which is not alpha-numeric.

11 FORWARD REFERENCE IN EQUATE OR ORG

The expression on the right side of an equals sign contains a symbol that has not been previously defined.

12 INVALID INDEX - MUST BE X OR Y

This error is given whether an indexed addressing mode is legal for the corresponding instruction or not. Following the operand expression is a comma and an invalid string where X or Y, denoting indexed addressing, was expected.

© 1976 PROGRAMMA CONSULTANTS

13 INVALID EXPRESSION

While evaluating an expression, the Assembler found a character that it could not interpret.

14 UNDEFINED ASSEMBLER DIRECTIVE

Either an invalid Assembler directive has been found, or the first three characters of one of the options in the .OPT directive are uninterpretable.

17 RELATIVE BRANCH OUT OF RANGE

A branch instruction can only branch either forward 127 bytes or backward 128 bytes relative to the beginning of the next instruction.

18 ILLEGAL OPERAND TYPE FOR THIS INSTRUCTION

The operand type found for the instruction in the operation field is not valid or not allowed.

19 OUT OF BOUNDS ON INDIRECT ADDRESSING

Since indirect addressing requires two bytes of page zero memory, the address referencing this area must be less than or equal to 254.

20 A, X, Y, S, AND P ARE RESERVED LABELS

The programmer has attempted to define one of the five reserved names as a symbol.

21 PROGRAM COUNTER NEGATIVE - RESET TO ZERO

Because there is no such thing as negative memory, an attempt to reference a negative position will cause this error message, and the program counter will be reset to zero.

© 1976 PROGRAMMA CONSULTANTS

APPLEPIE

A P P L E P I E VER 1.Ø

TEXT EDITOR SYSTEM

(c) 1979 PROGRAMMA INTL.

*** ALL RIGHTS RESERVED ***

"REPRODUCTION IN ANY PART OR FORM OF THE CONTENTS OF THIS DOCUMENT OR ITS ACCOMPANYING CASSETTE TAPE OR DISKETTE, EXCEPT FOR THE PERSONAL USE OF THE ORIGINAL PURCHASER, IS STRICTLY FORBIDDEN WITHOUT THE EXPRESSED WRITTEN CONSENT AND PERMISSION OF PROGRAMMA INTERNATIONAL."

PROGRAMMA

PRODUCT DESCRIPTION

PROGRAMMA INTERNATIONAL, INC.
3400 Wilshire Blvd.
Los Angeles, CA 90010

(213) 384-0579 • 384-1116 • 384-1117

TABLE OF CONTENTS

INTRODUCTION 1

EDITING COMMANDS 1

PIE MONITOR COMMANDS 5

INSTALLATION 6

PRINTER INTERFACE 6

UPPER & LOWER CASE 6

USING THE APPLEPIE TEXT EDITOR 7

ATTACHMENT I 14

1.Ø INTRODUCTION

PIE (PROGRAMMA Improved Editor) is a two-dimensional, cursor-based editor designed specifically for use with memory-mapped and cursor-addressed CRT's. It is totally different from the usual line-based editors, which were originally designed for Teletypes. PIE includes the following features:

- Blinking cursor
- Cursor movement up, down, right, left, plus tabs
- Character insert and delete
- String search forwards and backwards
- Page scrolling
- GOTO line number, plus top or bottom of file
- Line insert and delete anywhere on screen
- Move and copy (single and multiple lines)
- Append and clear to end of line
- Efficient memory usage

PIE cannot be used to create BASIC program tapes as the format is different.

2.Ø EDITING COMMANDS

All PIE editing commands consist of a single character, optionally preceeded by a [break] (referred to generically as [ARG]), followed by a numeric or string argument. The keyboard layout is shown in Attachment I.

Normally the editor is in a text insert mode. Any text entered from the keyboard will appear on the screen at the current cursor location. Therefore to distinguish commands from ordinary text, all alphabetic keys used as commands must be "controlled" (type the command key while the CONTROL key is depressed). Command names are all in upper case on the diagram, above the names of the appropriate key (i.e., CTRL-d is HOME).

© 1976 PROGRAMMA CONSULTANTS

The only command which does not require the CTRL key to be simultaneously depressed is (refer to Attachment I):

[CR]	Carriage return - left justify cursor on the next line (see note 2).

The following commands all require the CTRL key to be depressed simultaneously with the command key:

[LEFT]	Move cursor one position to the left
[RGHT]	Move cursor one position to the right
[UP]	Move cursor up one line
[DOWN]	Move cursor down one line
[HOME]	Home cursor in upper left hand corner
[BHOM]	Home cursor in lower left hand corner
[-PAG]	Move up (toward top of file) one "page" (20 lines)
[+PAG]	Move down (toward bottom of file) one page
[LTAB]	Move cursor left one horizontal tab (8 spaces)
[RTAB]	Move cursor right one horizontal tab
[GOTO]	Go to top of file (line 1)
[ARG]n[GOTO]	Go to line 'n' ($1 < n < 65536$). See notes 1 and 7
[KILL]	Delete the current line, unconditionally.
[-SCH]	Search backwards (up) into file for next occurence of string specified in last search command
[ARG]t[-SCH]	Search backwards for string 't'
[+SCH]	Search forwards (down) into the file for next occurence of string specified in last search. Search begins on line 2 of screen
[ARG]t[+SCH]	Search forward for string 't'
[APP]	Append - move cursor to last character of line, plus one

© 1976 PROGRAMMA CONSULTANTS

[ARG]n[KILL]	Delete 'n' lines (1<n<256), unconditionally.
[ARG]n[-PAG]	Move up 'n' "pages" toward top of file.
[ARG]n[+PAG]	Move down 'n' "pages" toward bottom of file.
[INS]	Insert one blank line before the current line
[ARG]n[INS]	Insert 'n' blank lines before the current line (1<n<256)
[DEL]	Delete the current line, saving it in the "push" buffer (see note 5)
[ARG]n[DEL]	Delete 'n' lines (1<n<256), saving the first 2Ø in the "push" buffer
[DBLK]	Delete current line as long as it is blank (closes up areas)
[PUSH]	Save current line in "push" buffer (see note 5)
[ARG]n[PUSH]	Save 'n' lines in the "push" buffer (1<n<21)
[POP]	Copy the contents of the "push" buffer before the current line
[CINS]	Enable character insert mode (see note 6)
[CINS][CINS]	Turn off character insert mode
[BS]	Backspace - delete the character to the left of the current cursor position. In insert mode, also pulls character to right of cursor left one position
[GOB]	Gobble - delete the current character and pull remainder of characters to right of cursor left one position
[EXIT]	Scroll all text off the screen and exit the editor

The following keys, which have already been defined above, have a different function if "controlled":

[ARG][HOME]	Home line - scroll up to move current line to top of screen
[APP][APP]	Left justify cursor on current line
[ARG][GOB]	Clear to EOL - delete from cursor to right side of screen
[ARG][GOTO]	Go to bottom of file (last line + 1)

© 1976 PROGRAMMA CONSULTANTS

Notes:

1) Lines are numbered starting with 1, 2, ... n. When an existing file is loaded it is renumbered. Line numbers are not displayed on the screen since they are not needed for any editing commands except [GOTO].

2) [CR] is a cursor motion only and does not put a carriage return on the screen. Use [ARG][GOB] to delete to the end of a line if necessary.

3) A carriage return is always forced into the last column (64) of every line. Therefore no characters can be typed in that position.

4) If [+SCH] and [-SCH] find the string being searched for, they move the line containing the string to the top of the screen. [GOTO] also moves the target line to the top of the screen. (Use [UP] to move the line down if desired). [BOT] moves all text off the screen.

5) The [DEL] and [PUSH] commands both save up to 2Ø lines of text in a special "push" buffer, which can be retrieved using the [POP] command. (Any text previously saved in the buffer is overwritten by a [DEL] or [PUSH] command.)

   The combination of [DEL] and [POP] therefore moves a line (or block of lines); and the combination of [PUSH] and [POP] copies a line (or block of lines). [POP] can be used more than once in a row to retrieve the same line(s) of text at different locations in the file.

6) Normally characters on the screen are overwritten when a new character is typed.

   In the character insert mode, enabled by [CINS], characters to the right of the cursor (plus the characters at the current cursor position) are pushed over to the right before the new character is entered on the screen.

7) When the [ARG] key is pressed, the system status line will display the word "ARG:" to accept a numeric or string argument. The only keys that are allowed in this mode are any printable characters, [BS], and [ARG]. The latter clears the system status line allowing input to start over. Any other function key will exit the argument mode and execute the command.

© 1976 PROGRAMMA CONSULTANTS

3.Ø PIE MONITOR COMMANDS

The following commands are used to load and write tape, load and save to disk (if the disk version is being used), and to edit and re-edit text. All commands are terminated by the RETURN key.

EDIT E	Edit a new file. The edit buffer and screen are cleared and the cursor is homed on line 1. Exit via [EXIT] key.
RE-EDIT R	Re-edit an existing file already in the edit buffer. The top line of the file will be on line 1. NOTE: Do NOT use the re-edit command unless there is text already in the edit buffer.
LOAD L	Load from cassette tape into the edit buffer a file of text. Use the re-edit command to edit the file.
SAVE S	Write the edit buffer out to cassette tape. The system cassette recorder must be running before the RETURN.
TYPE T	Type the contents of the edit buffer to a printer device that contains an Apple type of printer interface.
QUIT Q	Quit the editor command mode and return to the Apple II monitor. Same as a RESET.

+++ DISK COMMAND EXTENSION +++ DISK VERSION ONLY.

GET G	Get from diskette and into the edit buffer the file that the system will prompt for. Use the "R" command to re-edit the file.
PUT P	Write the file in the edit buffer to diskette. The system will prompt for the file name.
CATALOG C	Display the CATALOG of the diskette currently mounted on the active drive.
CTRL D	Will allow the execution of any Apple II DOS Disk command that follows the CTRL D itself.

© 1976 PROGRAMMA CONSULTANTS

4.Ø INSTALLATION

System requirements: Apple II, 16K, Cassette
Apple II, 32K, Disk
Cold start address: 8ØØH

To load the editor from cassette, first type 2ØØ.2ØØØR, while in the Apple II Monitor mode "*". When it has finished loading, the system will begin execution automatically.

To load the editor from diskette, boot the ApplePIE System diskette, or BRUN the file APPLE PIE.

5.Ø PRINTER INTERFACE

The ApplePIE System that is released will function with any printer that contains a standard APPLE II type interface. Further, the printer must be located on SLOT # 2.

In the event that the user wishes to interface his own type of printer, or to change the location of the supplied SLOT number, then the following location is modified to contain the address of the printer driver routine.

8Ø7/8Ø8H Printer driver address (pre-set to ØØ/C2)

On non-standard printer driver routines, the printer driver is responsible for setting the proper hook into the character output vector on page zero.

6.Ø UPPER & LOWER CASE

The present ApplePIE System assumes that all alphabetic characters (A-Z) are in lower case, unless a specific character is preceded by the character "]" (Shift M). All characters will appear to be in upper case when displayed on the CRT device, however, when output to the printer is desired, each character will be appropriately converted to upper or lower case. The "]" character is never printed. The appearance of the "]" character preceding any other character causes the character to be printed in upper case.

© 1976 PROGRAMMA CONSULTANTS

7. USING THE APPLEPIE TEXT EDITOR

ApplePIE is a "window editor" that utilizes the hardware features of the Apple II computer to perform part of the editing function. To invoke the ApplePIE text editor, the user enters "EDIT", while under the command mode. Note that the ApplePIE Text Editor prompts the user with the prompt "COMMAND?", while under the command mode. The "EDIT" command assumes that there exists no text in the edit buffer; hence, the editor is initialized to create a new file of text material. Upon entering the "EDIT" command, the following will be displayed on the CRT:

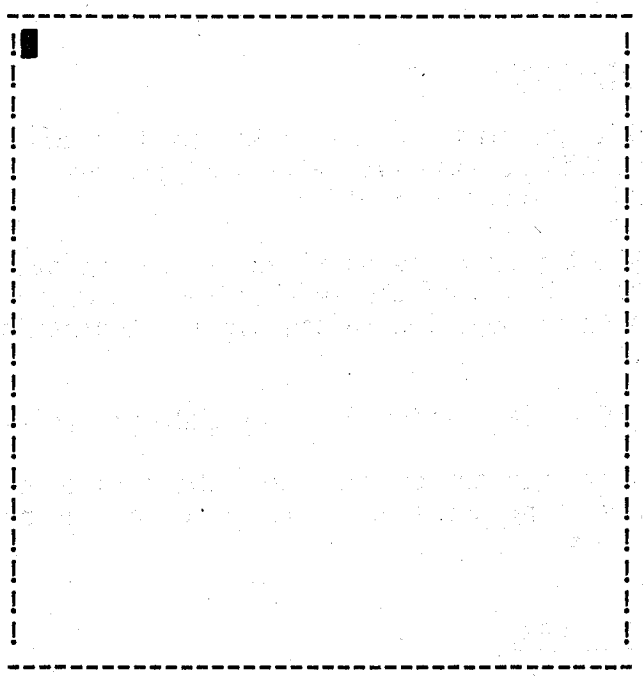


FIGURE 1

The above rectangle represents the "window" which will be used to enter and view text in the edit buffer. Note that the "EDIT" command caused the cursor to be placed at the "home" position on line # 1.

The window represents twenty (20) lines of sixty-four (64) characters each. Since the maximum number of characters that the window can hold across at any time is thirty-eight (38), then the representation of a sixty-four character line must be done using two (2) separate and unique windows. Figure 1 above represents the "left window".

© 1976 PROGRAMMA CONSULTANTS

The "window" display of the ApplePIE Text Editor consists of five parts:

1. The Window Border
2. The Left Window
3. The Right Window
4. The Data Area
5. The Argument/Message Line

Let us assume that an existing FORTRAN text file is to be edited. Figure 2 represents each of the above parts. Note that the columnar display is not displayed, and it is there to assist the user is this explaination.

```
         1         2         3          3         4         5         6
1234567890123456789012345678901234567  7890123456789012345678901234567890123
-----------------------------------------  ---------------------------------------
!      DIMENSION WF(20,20),XF(20),YF(20> <XF(20),YF(20),ZF(20)                  !
!      DATA I1/20/,JJ/20/,XK/27.4/     ! <                                     !
!      DO 20 I=1,I1                    ! <                                     !
!      DO 10 J=1,JJ                    ! <                                     !
!      WF(1,J)=0.0                     ! <                                     !
!   10 CONTINUE                        ! <                                     !
!   20 CONTINUE                        ! <                                     !
!      YK=32.4                         ! <                                     !
!C     THIS IS THE COMMAND USED TO READ> <USED TO READ THE INPUT                !
!      DO 40 I=1,I1                    ! <                                     !
!      READ(5,30) X1,X2                ! <                                     !
!   30 FORMAT(2F10.2)                  ! <                                     !
!      XF(I)=X1+XK                     ! <                                     !
!      YF(I)=X2+XK                     ! <                                     !
!   40 CONTINUE                        ! <                                     !
!      DO 50 I=1,I1                    ! <                                     !
!      ZF(I)=XF(I)*2.+YF(I)*3.         ! <                                     !
!   50 CONTINUE                        ! <                                     !
!      DO 90 I=1,I1                    ! <                                     !
!      IF(ZF(I)) 60,70,80              ! <                                     !
-----------------------------------------  ---------------------------------------
 ARG:  (ARGUMENT/MESSAGE LINE)              ARG:  (ARGUMENT/MESSAGE LINE)
```

FIGURE 2

The bottom line of the screen is reserved for messages that may be sent by the system to the user. PIE erases the bottom line whenever a new control command is entered. The bottom line is also used to enter any arguments that may be used in conjunction with any of the control keys.

© 1976 PROGRAMMA CONSULTANTS

The window border boxes the data area and informs the user if any line contains more than 38 characters. Note that any line longer than 38 characters will contain a > "greater than sign" on the right side border of the left window. When the user is viewing the right window, the left side border of the right window contains < "less than signs". This is an indication that the right window is being viewed/edited.

The data area starts immediately after the top border of "dashes" and continues to the end, before the bottom border of "dashes".

To enter line # 1 of Figure 2 the user enters:

ØØØØØØDIMENSION WF(20,20),XF(20),YF(20),ZF(20)

where Ø is a space (depress the space bar)

Note that the line that was just entered is longer than 38 characters, and when the 39th character was entered, the PIE system heuristically switched to the right window automatically. This allows the user to enter a line longer than 38 characters long. When the "CR" carriage return was entered, the PIE system returned the display to the left window and positioned the cursor on line # 2 of the left window display.

Lines # 2 through 20 are entered in the same fashion as line # 1 above. Note that several lines contained less than 38 characters, therefore, the PIE system never needed to switch to the right window display. In Figure 2, the system will only switch to the right window display for lines # 1 and 9 only.

Lets assumes that we are about to enter line # 20 of Figure 2 and the following text is entered before executing the "CR":

ØØØØØØIF(ZF(I)) 60,ZZ,80

The characters "ZZ" are incorrect and we need to correct them to reflect the characters "70". The simplest form of correction is to use the backspace "BS" command (the left-arrow ←) key and backspace to the first "Z". Note that the "BS" command will delete any characters that are backspaced through. Hence it will be necessary to re-type the comma after the "70" followed by the "80".

Lets assume that we would like to change the word "INPUT" on line # 9 to "DATA". First we must realize that this line contains more than 38 characters, and further, the word "INPUT" exists on the right window. To switch the display to the right window, we must first position the cursor on any line that may contain more than 38 characters. This is quite simple... look for "greater than signs" on the border display.

© 1976 PROGRAMMA CONSULTANTS

To switch the display of Figure 2 to the right window, we must move the cursor from line # 20 to line # 9. There are several ways of accomplishing the above, however, we will introduce the general method of PIE cursor positioning. To move the cursor toward the top the command "UP" (CTRL-E) key is depressed to move the cursor whatever number of times is necessary for the desired result. Once we have positioned ourselves on line # 9, we must switch to the right window display. This is accomplished by using the append command "APP" (CTRL-U or right-arror) key.

Once the display switches to the right window, we can position the cursor on top of the "I" of "INPUT" by using the appropriate cursor movement commands. To change "INPUT" to "DATA" we simply type "DATAØ" on top of the word "INPUT". Note that by depressing the "APP" command key we can cause instanteneous switching from the left window to the right window and vice-versa.

Now, lets position the cursor on line # 20:

Enter the "BHOM" command key (CTRL-X)

The cursor should be on line # 20 position # 1. Lets do a CR and see what happens... the window scrolled and we have a new blank line at line # 20. This has now caused the window to be positioned one line after the first line of text in the text edit buffer. Lets type the following text line:

THE CAT RAN OVER THE LAZY DOG'S BACK

Note that the screen once again scrolled and we have a new line at line # 20 of the left window.

Lets position the cursor at the "home" position on the right window:

Enter the "HOME" command key (CTRL-D)

Now lets take a look at the text that exists on top of the edit buffer. We can cause the window to be moved toward the top of the edit buffer in several ways. First, we can enter the "UP" command key and the text will move one line down. Subsequent depressions of the "UP" key will cause text lines to move down, until the top or first line of text in the text edit buffer is reached. Another method of moving the window toward the top is to scroll up. The "-PAG" command key (CTRL-R) causes a page (20 lines) of text to be filled on the window. Actually, the window is moved back by 20 lines, if possible. A final way of going to the top of the text edit buffer is by using the "GOTO" command key. Here "GOTO" (CTRL-T) causes the system to position the window on the top line of text in the text edit buffer (or line # 1). Note that while executing most of the commands above, the ARGUMENT/MESSAGE line contained the message ****BUSY****.

© 1976 PROGRAMMA CONSULTANTS

Now, position the cursor on the line that contains the following text of Figure 2:

```
DIMENSION WF(20,20),XF(20),YF(20),ZF(20)
```

We can copy the contents of that line and transfer it to an internal buffer called the "push-down buffer". The "PUSH" command key (CTRL-L) accomplishes the above task.

Now, position the cursor on the line that contains the following text:

```
40 CONTINUE
```

Lets make a copy of the line that was previously saved on the "push-down buffer" directly below the above line:

Enter the "POP" command key (CTRL-O)

Note that the line that was originally in the "push-down buffer" is now copied directly below the line where the cursor was positioned. This is the method that is used to copy lines in the ApplePIE Text Editor.

Now, lets make several copies of the same line:

Enter the "POP" command key several times.

Note that the line that is in the "push-down buffer" gets copied as many times as you depressed the "POP" command key. Let us now delete all of the lines that we inserted using the "POP" command. Position the cursor on the first instance of the inserted line.

Enter the "KILL" command key to delete a line (CTRL-N).

Enter as many "KILL" commands to delete all of the lines that were previously inserted through the "POP" command.

Scroll to the top of the text file by using the "GOTO" command key (CTRL-T). Position the cursor on the following line:

```
20 CONTINUE
```

Lets insert a blank line following the line where the cursor is presently positioned by entering a "INS" command key (CTRL-I). Insert several blank lines by depressing the "INS" command key numeous times. Note how the text that followes the blank lines is automatically scrolled below the screen.

Now position the cursor on the first instance of a blank line that was inserted by using the "INS" command key.

© 1976 PROGRAMMA CONSULTANTS

Lets assume that we would like to delete all blank lines that exist between two lines of source text. We can use the "DBLK" command key to compress the text (CTRL-B).

Note that all blank lines that were previously inserted through the "INS" command key no longer exist.

Now position the cursor on the following line of Figure 2:

```
C     THIS IS THE COMMAND USED TO READ DATA
```

Switch the display to the right window by using the "APP" command key. Position the cursor on top of the "D" of "DATA" by using the "LEFT" command key (CTRL-S). Lets turn-on the character insert mode available in the ApplePIE system to insert the following text:

```
/INPUT
```

Note that the ARGUMENT/MESSAGE LINE contains the following message to alert you of being in ****I N S E R T M O D E****, once you enter the "CINS" command key (CTRL-P). Now enter the "/INPUT" characters. Note that the "DATA" characters were automatically moved to the right. Now turn-off the character insert mode by entering "CINS" once again. Note that this command executes as a toggle.

Lets position the cursor on top of the "/" of the current line. Assume we want to delete the characters that we had just entered... enter the "GOB" command key (CTRL-J) to delete the present character. Note that characters to the right of the cursor are automatically shifted to the left by the "GOB" command key.

Now, lets delete the current line and at the same time save it to the push-down buffer. Enter the "DEL" command key (CTRL-K) to move a line from the window and into the push-down buffer. If you want to bring the line back, then "POP" the push-down buffer. This is the manner in which lines can be moved in the ApplePIE Text Editor.

So far in this edit session we have edited text using most of the command keys that are available in the ApplePIE Text Editor. Note that most editing occurred on a single line basis. However, at times it becomes very useful to be able to move or copy several lines of text, scroll through several pages, GOTO a particular logical line number, insert/delete several lines. The "ARG" command key (ESC) is used to prompt the user for an argument (parameter) that will be used in the processing of the next immediate command key. The system assumes that if no argument is given, then one (1) will be the argument. Hence when we performed the "INS", one line was inserted; or when the "DEL" was done, one line was deleted; etc.

© 1976 PROGRAMMA CONSULTANTS

To GOTO logical line number 22 the following sequence would be entered:

Depress the "ARG" command key (ESC)-
Enter the characters 22
Depress the "GOTO" command key (CTRL-T)

Most of the commands in the ApplePIE Text Editor will function with some type of argument. The user is instructed to refer the documentation and description of the function of the command keys.

One final point of interest is the ability of the ApplePIE Text Editor to search through the text edit buffer for any particular text string. The argument line is used to supply the system with the text string that is desired in the text.

Assume that we would like to search for the string "IF" in Figure 2. Lets first position ourselves at the top of the text buffer by doing a GOTO command key with no argument.

Enter ESC and the string "IF"
Enter the "+SCH" command key (CTRL-Z)

The "+SCH" will cause the system to perform a search for the first instance of the argument string. If found, the cursor is positioned on the string itself. If not found, the system will respond with an appropriate message on the ARGUMENT/MESSAGE line.

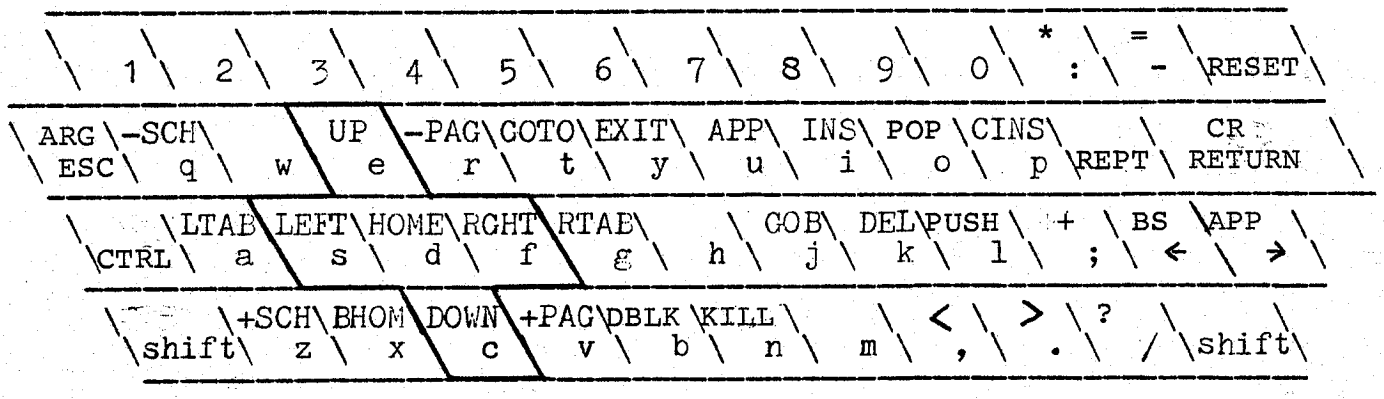
To search for a second instance of the string, without changing the argument, the user then depresses the "+SCH" command key again. The search proceeds from the point where the cursor was last positioned. Note that the "ARG" did not have to be specified because it did not change. The "-SCH" command key will perform a search toward the top of the text, relative to the present position of the window.

Finally, you have now completely finished your text editing job and would like to return to the ApplePIE Text Processor "COMMAND" mode. The "EXIT" command key (CTRL-Y) is used to accomplish the above. Should you need to re-edit text once again, you may return to the ApplePIE Text Editor by entering the "RE-EDIT" command, while in the COMMAND mode.

© 1976 PROGRAMMA CONSULTANTS

ATTACHMENT I

KEYBOARD LAYOUT


1 2 3 4 5 6 7 8 9 0 * = :
- RESET
ARG ESC -SCH q w UP e -PAG r COTO t EXIT y APP u INS i POP o CINS p REPT
CR RETURN
LTAB CTRL a LEFT s HOME d RGHT f RTAB g GOB h DEL j PUSH k + l ; BS ←
APP →
+SCH shift z BHOM x DOWN c +PAG v DBLK b KILL n m < , > . ? /
shift